Proyecto fin de carrera

Ingeniero Técnico en Informática de Sistemas

MODIFICACIÓN DEL PLANIFICADOR DE LINUX PARA EXTRAER INFORMACIÓN DE RENDIMIENTO DE LOS PROCESOS

Jorge Pastor Pérez

Dirigido por Vicent Lorente Garcés
Departamento de Informática de Sistemas y Computadores

Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Valencia, septiembre de 2011

Índice de contenido

Resumen....4
1 Introducción....5
2 Estado del arte....7
2.1 Scheduling Algorithms with Bus Bandwidth Considerations for SMPs....7
2.2 On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling....12
3 Trabajo relacionado....16
3.1 Arquitectura de un procesador....16
3.1.1 Introducción a la microarquitectura Core....16
3.1.2 La jerarquía de memoria....16
3.1.3 La unidad de monitorización del rendimiento....18
3.2 Los contadores de rendimiento en el sistema operativo....22
3.2.1 La llamada al sistema sys_perf_event_open()....22
3.3 Benchmarks....28
3.3.1 SPEC CPU2006....28
4 Experimentos y datos....32
4.1 Desarrollo de una herramienta para la toma de datos....32
4.1.1 Diseño....32
4.1.2 Lanzamiento de las aplicaciones....33
4.1.3 Recogida de datos de los benchmarks....33
4.1.4 Selección y conversión de los datos....38
4.1.5 Representación de la información....40
4.2 Realización de las pruebas de rendimiento....42
4.2.1 Configuración del entorno de ejecución....42
4.2.2 Resultados de la ejecución de los benchmarks....42
5 Conclusiones y trabajo futuro....57
Bibliografía....58

Resumen

En los últimos tiempos, los sistemas multiprocesador han proliferado. En los sistemas multiprocesador, diferentes núcleos del procesador compiten por el acceso al bus de memoria cuando deben acceder a datos que no están disponibles en memoria caché, debido a que el acceso a memoria principal se realiza en serie. Esto es un importante cuello de botella arquitectural que puede hacer que el rendimiento se vea afectado.

El planificador de procesos es un elemento esencial en los sistemas operativos multiprogramados, el cual decide qué proceso debe pasar a CPU de entre los que se encuentran preparados para ejecución. En los planificadores de tiempo compartido actuales, se planifica por prioridad y afinidad de los procesos. La afinidad de los procesos trata de evitar dentro de lo posible las migraciones de procesos entre unidades de ejecución de la CPU para mantener el estado que el proceso tiene en la caché y no tener que afrontar el coste que supone generarlo. Sin embargo, los planificadores actuales no abordan directamente las limitaciones del ancho de banda del bus. Como puede verse los estudios anteriores [1] y [2], planificar teniendo en cuenta el ancho de banda del bus de la memoria puede significar una mejora en el rendimiento de los sistemas multiprocesador.

En los procesadores modernos, existen contadores de monitorización del rendimiento, que permiten obtener información precisa de cómo se comporta el sistema. Los sistemas operativos actuales permiten tener acceso a estos contadores siempre y cuando sean ofrecidos por la arquitectura hardware. Basándose en estos datos, es posible caracterizar el comportamiento de las aplicaciones a lo largo del tiempo, y guiar a un planificador de procesos de modo que planifique teniendo en cuenta las limitaciones del ancho de banda del bus. Para ello, previamente es necesario comprender el funcionamiento de los contadores y diseñar métodos para recoger los datos, tratarlos e interpretarlos. Todo esto es posible con la ayuda de programas de prueba llamados *benchmarks*, que se comportan de forma predecible y permiten comparar los resultados entre ejecuciones.

1 Introducción

Los multiprocesadores de memoria compartida, como los recientes procesadores multinúcleo, son ampliamente utilizados como una vía asequible para construir desde sistemas de alto rendimiento hasta equipos de escritorio o incluso dispositivos móviles. El bus que interconecta a los procesadores con la memoria principal ha llegado a ser el cuello de botella principal que dificulta la escalabilidad de estos sistemas. Debido a esto, en los últimos años se han realizado diversos estudios y se han propuesto algunas políticas de planificación conscientes del ancho de banda del bus de memoria que tratan de mitigar el problema.

Los planificadores actuales utilizados en sistemas con multiprocesadores de memoria compartida emplean tiempo compartido con prioridades dinámicas y afinidad de los procesos. La afinidad de los procesos tiene como objetivo intentar planificar siempre en el mismo procesador aquellos procesos que hayan conseguido construir su estado en la caché, para evitar afrontar el coste que supone generarlo. Algunos planificadores utilizan algoritmos de espacio compartido, que dividen los procesadores entre programas, de modo que cada uno de ellos se ejecute en un subconjunto de los procesadores del sistema, cuyo tamaño puede variar en tiempo de ejecución. Gracias a la mejora de la localidad, se tiende a aumentar el rendimiento de la caché en tareas paralelas. Sin embargo, se limita el grado de paralelismo que la aplicación puede aprovechar.

Ciertos estudios muestran la importancia del ancho de banda de memoria para el rendimiento de los sistemas con multiprocesadores de memoria compartida, y plantean políticas de planificación alternativas. En las políticas de planificación según el consumo de ancho de banda del bus de memoria, se busca planificar tareas de modo que se realice una utilización del bus de memoria inteligente, que ni lo sature ni lo infrautilice.

Para poder guiar a un planificador en sus decisiones dependiendo del consumo de ancho de banda de memoria del bus, hay que predecir cuáles van a ser los requisitos de una aplicación, tarea o hilo. Si se quiere conocer el comportamiento de una aplicación, hay que caracterizarla correctamente.

Con el fin de caracterizar las aplicaciones es necesario emplear los contadores de rendimiento disponibles en las CPUs actuales. Estos contadores permiten registrar diferentes tipos de eventos, dependiendo de la microarquitectura del procesador. Los eventos pueden ser utilizados para caracterizar las aplicaciones en cuanto a accesos a memoria o utilización del ancho de banda del bus de la memoria, entre otras cosas. Para poder medir correctamente con los contadores de rendimiento, también es necesario contar con la ayuda de un sistema operativo debidamente capacitado.

Un planificador orientado al consumo de memoria podrá tomar las decisiones en cada cambio de contexto. Para ello, hay dos posibles escenarios, dependiendo de las necesidades. Por un lado, es posible realizar previamente la caracterización de las tareas para guiar posteriormente al planificador. Esto tiene la ventaja de guiar al planificador con el comportamiento preciso de las aplicaciones, pero tiene un uso muy limitado. Por otro lado, existe la posibilidad de guiar al planificador según el comportamiento en tiempo real de las aplicaciones, tomando para ello una ventana de muestras antes del instante actual. En cualquier caso, es necesario medir con los contadores de rendimiento.

Para poder evaluar la herramienta de medida implementada en el proyecto, se utilizan *benchmarks*. Los *benchmarks* proporcionan un entorno de ejecución predecible, el cual puede reproducirse tantas veces como sea necesario. De este modo, es posible analizar los resultados tras

realizar cambios en la herramienta o el entorno de ejecución, permitiendo extraer conclusiones y efectuar ajustes oportunos.

El resto del proyecto está organizado como sigue: en el punto 2, se habla de algunos estudios existentes relacionados con la saturación del bus de la memoria y políticas de planificación en sistemas multiprocesador. En el punto 3 se detalla el trabajo relacionado, como la arquitectura de la CPU, el sistema operativo o las pruebas de rendimiento. A continuación se pasa en el punto 4 a explicar los procedimientos empleados y los resultados de las pruebas realizadas. Por último, en el punto 5, se encuentran las conclusiones y el trabajo futuro.

2 Estado del arte

2.1 Scheduling Algorithms with Bus Bandwidth Considerations for SMPs

En [1], C. D. Antonopoulos et al. observan que en multiproceso simétrico ninguno de los algoritmos de planificación anteriormente propuestos ha sido impulsado por los efectos de compartir otros recursos del sistema que no fueran las cachés y los procesadores. En concreto, ninguna de las políticas está impulsada por el impacto de la compartición del bus, o en general, de la red que interconecta los procesadores y la memoria. Además, de entre todas las políticas que se centran en optimizar el rendimiento de la memoria, ninguna considera el ancho de banda disponible entre los diferentes niveles de la jerarquía de memoria como un factor para orientar las decisiones de planificación. Tras esta reflexión, los autores efectúan un estudio de las implicaciones que tiene el ancho de banda del bus en el rendimiento de las aplicaciones.

Para realizar las experiencias, se utilizan aplicaciones optimizadas de dos *benchmarks*, *NAS* y *Splash-2*. La plataforma experimental consiste en un sistema con 4 procesadores simétricos, cada uno de los cuales tiene su propia memoria caché de segundo nivel. El sistema operativo es un Linux de la rama 2.4. La tasa de transferencia máxima sostenida en el bus se estima en 29'5 transacciones/µs.

Las pruebas se dividen en cuatro grupos. El primero mide el ancho de banda consumido por cada aplicación, cuando se ejecuta en solitario utilizando dos procesadores. En el segundo grupo, se ejecutan dos instancias idénticas de una aplicación, utilizando dos procesadores cada una.

En el tercer grupo, se ejecuta una instancia de la aplicación que utiliza dos procesadores, junto a dos instancias de un *microbenchmark BBMA*. El *microbenchmark BBMA* está diseñado de modo que obtenga una tasa de acierto cercana al 0% y realice constantemente accesos a memoria, consumiendo una fracción significativa del ancho de banda del bus disponible.

El cuarto grupo es idéntico al tercero, excepto por la configuración del *microbenchmark*. En este caso se utiliza un *microbenchmark nBBMA*, que está diseñado para obtener una tasa de acierto cercana al 100%, a pesar de los constantes accesos a memoria, generando muy poco tráfico en el bus.

En las barras negras de la figura 1 se muestra el ancho de banda del bus consumido por cada aplicación (primer grupo de pruebas). Dado que el ancho de banda máximo consumido es de 23'31 transacciones/µs, en este caso el bus ofrece suficiente ancho de banda para las aplicaciones en solitario.

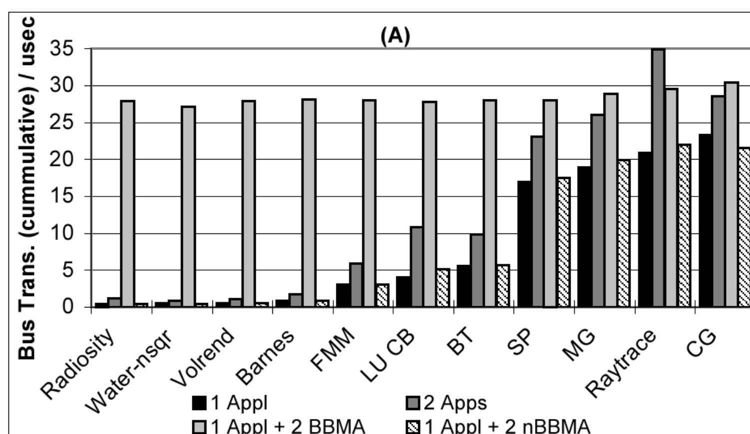

Figura 1: Tasa de transferencia del bus acumulada

Las barras en gris oscuro representan al segundo grupo. Las aplicaciones con los mayores requisitos de ancho de banda ponen al bus cerca del límite. En la figura 2 se observa la pérdida de rendimiento sufrida por las aplicaciones, debido a la competición por el bus. Los resultados muestran que las aplicaciones con requisitos de ancho de banda altos sufren una mayor degradación.

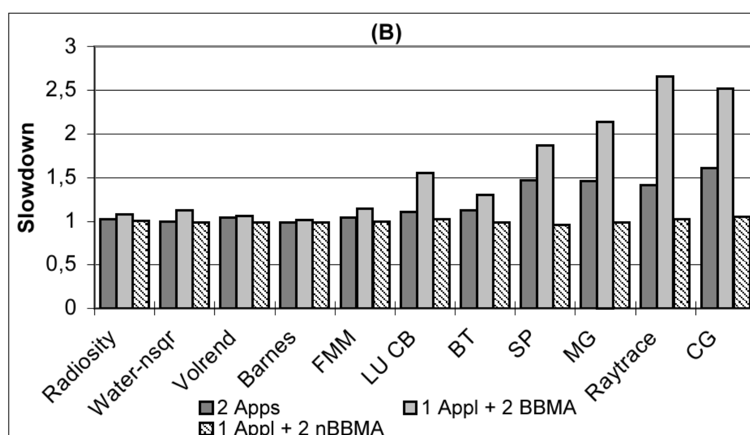

Figura 2: Pérdida de rendimiento

El tercer grupo está representado por las barras gris claro. Estos experimentos muestran el impacto de tener aplicaciones funcionando en un bus ya saturado. El ancho de banda consumido por cada carga de trabajo se encuentra cerca del límite de saturación. Las aplicaciones que realizan accesos intensivos a memoria sufren importantes pérdidas de rendimiento en este escenario.

Por último, las barras blancas rayadas se corresponden con el cuarto grupo. Tanto la tasa de transferencia de datos en el bus como el tiempo de ejecución de las aplicaciones son casi idénticos a los del primer grupo. Esto indica que combinar aplicaciones con altos y bajos requisitos de ancho de banda, es un buen método para alcanzar mayores tasas de rendimiento.

Los resultados obtenidos les motivan para investigar nuevas políticas de planificación que sean guiadas por el consumo del ancho de banda del bus. En total son implementadas dos nuevas políticas de planificación, las cuales planifican las tareas teniendo en cuenta el ancho de banda que consumen. El objetivo es optimizar el uso del ancho de banda del bus del sistema, planificando tareas que ni infrautilicen ni saturen el bus. La información requerida para guiar a las políticas de

planificación se obtiene de los contadores de monitorización del rendimiento presentes en los procesadores modernos. Estos contadores permiten realizar un análisis de los cuellos de botella que afectan al rendimiento. La información, que es posible obtener de estos contadores, no se había utilizado previamente en tiempo de ejecución para influir en las decisiones de planificación de un sistema real.

La primera política de planificación implementada se conoce como *Latest Quantum*. Al final de cada *quantum* del planificador, esta política actualiza las estadísticas de consumo de ancho de banda del bus para todas las tareas en ejecución, utilizando la información proporcionada por las aplicaciones. Las aplicaciones que anteriormente habían estado en ejecución, pasan al final de la lista. A continuación, el planificador selecciona las aplicaciones que serán ejecutadas en el siguiente *quantum*. Para evitar la inanición, se selecciona por defecto la primera aplicación de la lista. Cada vez que se selecciona una aplicación para ejecución, el ancho de banda del bus disponible en el sistema se calcula restando al ancho de banda total del bus del sistema los requisitos de todas las aplicaciones que ya habían sido asignadas. Tan pronto como hay procesadores disponibles, el planificador atraviesa la lista de aplicaciones. Para cada aplicación que encaje en los procesadores disponibles, se calcula un valor de conveniencia mediante los datos anteriores. Al finalizar el recorrido de la lista, se selecciona la aplicación más conveniente para ser ejecutada durante el siguiente *quantum*. Esta selección favorece el aprovechamiento óptimo del bus.

La segunda política de planificación se conoce como *Quanta Window*. La única diferencia con *Latest Quantum* es que en lugar de tener en cuenta los requisitos de ancho de banda del bus de cada hilo durante el último *quantum* del planificador, se utiliza la media de sus requisitos durante un limitado número de muestras anteriores. Esto tiene el efecto de suavizar los cambios repentinos en las transacciones del bus causados por una aplicación, filtrando ráfagas de corta duración u otras ráfagas causadas por eventos externos puntuales. Por otra parte, tiene la desventaja de reducir el tiempo de respuesta debido a cambios reales en el consumo de ancho de banda del bus.

Para evaluar la efectividad de estas políticas, se experimenta con tres conjuntos de cargas de trabajo heterogéneas.

El primer conjunto (figura 3) está formado por dos instancias de una aplicación de interés, cada una de las cuales requiere de dos procesadores, que se ejecutan junto a cuatro instancias del *microbenchmark BBMA*. En este conjunto, aplicaciones de interés coexisten con *microbenchmarks* que tienen una alta demanda del bus, permitiendo evaluar la eficiencia de las políticas en un bus ya saturado. *Latest Quantum* alcanza mejoras en el tiempo de respuesta que van desde un 4% hasta un 68%, obtieniendo una media del 41%. Las mejoras introducidas por *Quanta Window* varían entre un 2% y un 53%, con una media de un 31%.

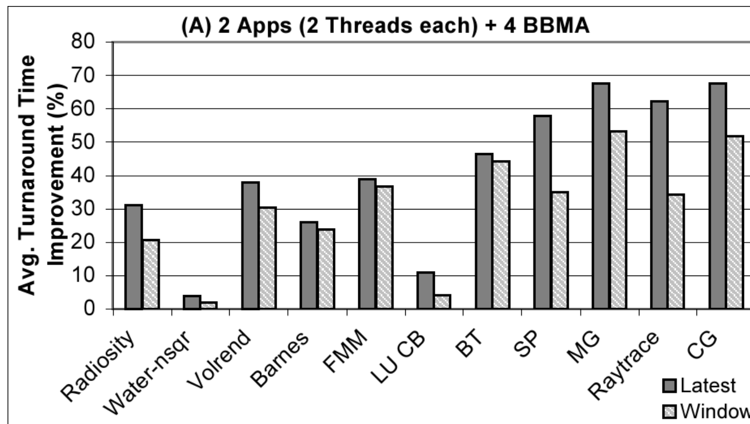

Figura 3: Mejora de rendimiento con respecto al planificador nativo (conjunto de aplicaciones A)

El segundo conjunto (figura 4) está formado por dos instancias de la aplicación de interés (requiriendo dos procesadores cada una) y cuatro instancias del *microbenchmark nBBMA*. En este conjunto, aplicaciones de interés coexisten con *microbenchmarks* que no añaden sobrecarga al bus. Este experimento demuestra la funcionalidad de las políticas propuestas cuando se encuentran disponibles en el sistema tareas con una baja demanda de ancho de banda. *Latest Quantum* alcanza hasta un 60% de mejora en el rendimiento, con una media del 13%. No obstante, hay tres aplicaciones que empeoran con respecto al planificador nativo del sistema operativo, siendo la reducción de hasta un 19%. Esto es debido a fluctuaciones irregulares en los requisitos de ancho de banda de estas aplicaciones, a las que la política *Latest Quantum* es sensible. Por otro lado, *Quanta Window* es mucho más estable, y sólo una aplicación sufre una degradación de un 1%, obteniéndose una mejora de hasta un 64%, y de media un 21%.

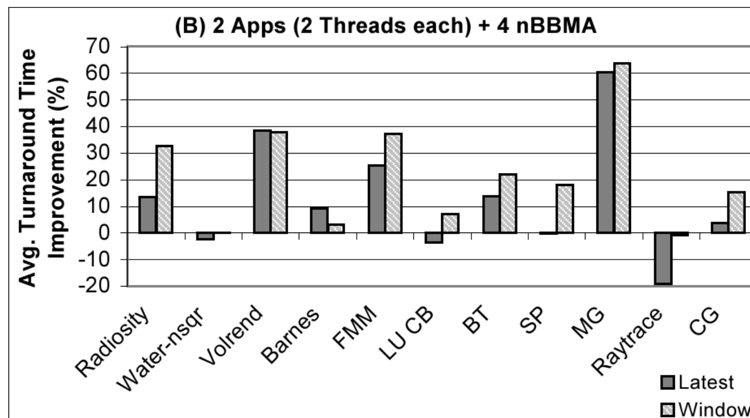

Figura 4: Mejora de rendimiento con respecto al planificador nativo (conjunto de aplicaciones B)

El tercer conjunto de cargas de trabajo (figura 5) combina dos instancias de la aplicación de interés (requiriendo dos procesadores cada una) con dos instancias del *microbenchmark BBMA* y dos instancias del *microbenchmark nBBMA*. Dichas cargas de trabajo simulan entornos de ejecución donde las aplicaciones de interés coexisten con otras que tienen una alta demanda de ancho de banda y una baja demanda. *Latest Quantum* ofrece una mejora media de un 26%, sin embargo, una

aplicación sufre una degradación del rendimiento de un 7%. En el caso de *Quanta Window*, la mejora media es de un 25%, mientras que dos aplicaciones sufren pérdida de rendimiento, siendo la degradación máxima de un 5%.

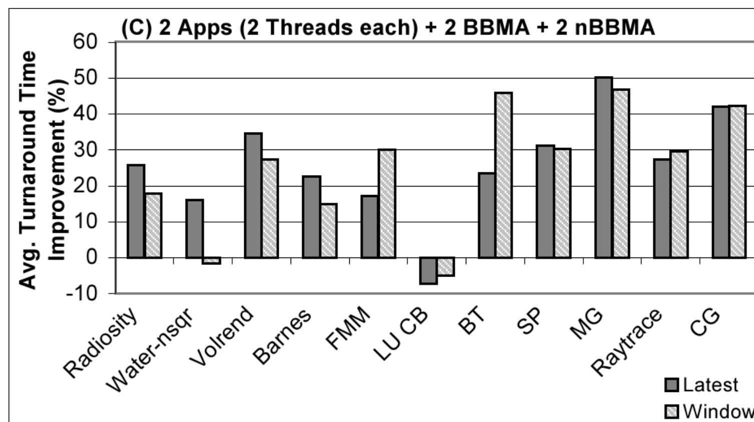

Figura 5: Mejora de rendimiento con respecto al planificador nativo (conjunto de aplicaciones C)

Con estos resultados, C. D. Antonopoulos et al. concluyen que ambas políticas ofrecen una mejora sobre el planificador nativo de Linux. *Quanta Window* se ha mostrado mucho más estable que *Latest Quantum*, manteniendo un buen rendimiento incluso en los casos extremos en los que *Latest Quantum* ha mostrado ser muy sensible a las peculiaridades de las aplicaciones.

Por tanto, la planificación de tareas consciente del ancho de banda es una estrategia efectiva cuando el número de tareas concurrentes supera al número de procesadores en un sistema multinúcleo. Se intenta predecir los requisitos de ancho de banda de una tarea, y seleccionar tareas concurrentes cuya suma de requisitos de ancho de banda eviten la saturación del bus, al mismo tiempo que mantienen una completa utilización del ancho de banda disponible. Sin embargo, hay algunos detalles en este enfoque que pueden afectar de forma importante al rendimiento del sistema y que no han sido tenidos en cuenta, habiendo sido posteriormente estudiados más a fondo.

2.2 On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling

En [2], Di Xu et al. observan que con los diseños de políticas de planificación anteriores, se pueden producir competiciones por el bus dentro del *quantum* de planificación, debido a fluctuaciones irregulares en la intensidad de los accesos a memoria.

En la mayoría de los esquemas de planificación conscientes del ancho de banda, los segmentos de tarea que son ejecutados durante un *quantum* del planificador son utilizados como unidades de planificación. Las políticas de planificación existentes intentan seleccionar y planificar segmentos de tarea que puedan maximizar sus requisitos totales de ancho de banda, de modo que se acerque lo máximo posible al ancho de banda de memoria que puede proporcionar el sistema. Esto está basado en la premisa de que estos segmentos de tareas podrían utilizar completamente el ancho de banda disponible en el sistema, y experimentar poca degradación del rendimiento antes de que se alcance el tope del del ancho de banda del sistema.

Sin embargo, se ha examinado esta premisa con un mayor nivel de detalle y se ha comprobado que casi nunca es el caso real. Es posible observar que los segmentos de tareas concurrentes sufren pérdidas de rendimiento significativas antes de que la combinación de sus requisitos de ancho de banda alcance el tope del ancho de banda. Otros análisis también muestran que los accesos a memoria aparecen de forma algo irregular durante la ejecución del programa. Para guiar su planificación, un planificador típico usa los requisitos de ancho de banda medio en un *quantum* de planificación. Por lo tanto, solamente es consciente de los cambios en los requisitos del ancho de banda entre los *quantum* de planificación, sin tener en cuenta las profundas fluctuaciones posibles dentro de los mismos. Por consiguiente, aún podrían suceder graves competiciones por el ancho de banda dentro de un *quantum* (incluso si su ancho de banda medio combinado es inferior al tope del ancho de banda) y degradar el rendimiento total, especialmente para cargas de trabajo con bajos requisitos de ancho de banda. En realidad, las fluctuaciones irregulares en los requisitos del ancho de banda son bastante comunes en aplicaciones reales.

Basándose en esto, Di Xu et al. diseñan una nueva estrategia de planificación, cuya base teórica es como sigue:

El tiempo de finalización ideal de una tarea es el tiempo mínimo que tardaría la tarea en completarse desde que fue enviada. Cuando una carga de trabajo se ejecuta en un sistema real, la competición por el ancho de banda provoca que el tiempo que tarda en completarse sea más largo que su tiempo de finalización ideal. La ejecución de una carga de trabajo ha de ser vista como el proceso de finalizar todas sus peticiones de memoria. Los planificadores conscientes del ancho de banda anteriores tratan de alcanzar una mayor utilización del ancho de banda, y a menudo tareas paralelas sufren una disminución del rendimiento lineal.

En el nuevo planificador, en cambio, se intenta planificar tareas cuyos requisitos de ancho de banda combinados permitan mantener el ancho de banda del bus en un nivel que minimice el impacto de la competición, incluso si todavía hubiera ancho de banda disponible en el sistema. Este nivel es llamado ancho de banda medio ideal de la carga de trabajo (*IABW*), y se calcula como la relación entre el número total de accesos a memoria y su tiempo de finalización ideal.

$$IABW = \frac{Número\ total\ de\ accesos\ a\ memoria}{Tiempo\ de\ completado\ ideal}$$

Las estimaciones teóricas realizadas corroboran que a la hora de planificar tareas, en lugar de tener como objetivo una utilización igual al tope del ancho de banda, se debería apostar por mantener los requisitos totales de ancho de banda en un nivel estable que equivalga al ancho de banda medio ideal de toda la carga de trabajo, siempre que sea posible. Ello podría minimizar la degradación del rendimiento debido a la competición.

No obstante, el *IABW* estimado no puede ser empleado como el valor de referencia de la política de planificación. Esto es debido a que la política de planificación propuesta está basada en los requisitos de ancho de banda cuando cada tarea se ejecuta en solitario. Sin embargo, en tiempo de ejecución, sólo es posible conocer el ancho de banda que cada tarea ha obtenido cuando compite con otras tareas concurrentes. El ancho de banda que cada tarea obtiene es normalmente menor (y a menudo mucho menor) que el observado cuando cada tarea se ejecuta en solitario, debido a competiciones por el bus de grano fino o incluso saturación del bus. Para poder mantener las mismas decisiones de planificación, un valor de referencia de ancho de banda realista debería ser también más pequeño que el *IABW* estimado. Este valor debe ser calculado.

Para realizar las experiencias que permitan determinar cómo calcular el ancho de banda de referencia óptimo para la planificación, se utilizan las pruebas de coma flotante del *SPEC CPU2006* y las pruebas del *NAS Parallel Benchmark (NPB)*. Primero se preparan 3 grupos de cargas de trabajo, cada una de las cuales estará compuesta por diferentes pruebas repartidas de forma pseudoaleatoria. Dos grupos estarán formados por cargas de trabajo con pruebas del *SPEC* y del *NPB*, respectivamente. El otro grupo contendrá cargas de trabajo piloto, formadas por pruebas del *SPEC*, y que se utilizarán para obtener el ancho de banda óptimo.

El cálculo del ancho de banda óptimo para cada carga de trabajo piloto, se realiza de la siguiente forma:

En primer lugar, se ejecuta la carga de trabajo partiendo de un ancho de banda de referencia igual al *IABW* y se mide el tiempo de finalización. Acto seguido se vuelve a ejecutar la carga de trabajo reduciendo el ancho de banda de referencia, observándose una mejora en el tiempo de finalización. Esto se sigue realizando paulatinamente hasta que la tendencia se invierta. Este proceso puede observarse en la figura 6.

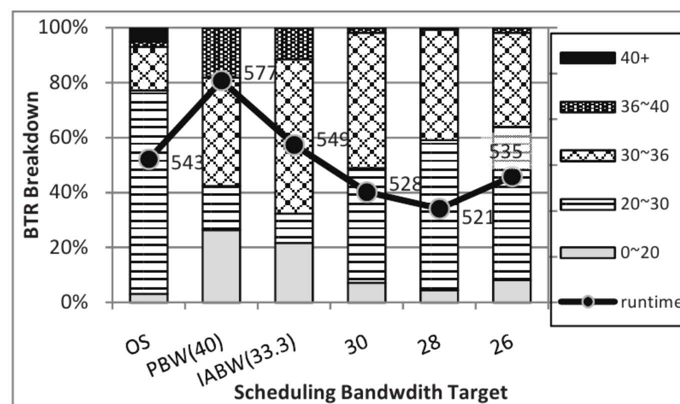

Figura 6: Ajuste del rendimiento de una carga de trabajo variando el ancho de banda de referencia

Una vez se ha ajustado el ancho de banda óptimo para cada una de las cargas de trabajo, se calcula la función que determina la relación existente entre el *IABW* y el ancho de banda de referencia para la planificación, por medio de un método de regresión polinómica.

$$Ancho\ de\ banda\ para\ la\ planificación = -0'0118 \times IABW^2 + 1'4571 \times IABW - 6'2403$$

La curva obtenida puede verse en la figura 7. Los cuadrados que aparecen representan la correspondencia entre el ancho de banda de referencia óptimo de cada una de las cargas de trabajo piloto (eje *y* izquierdo) y su *IABW* (eje *x*).

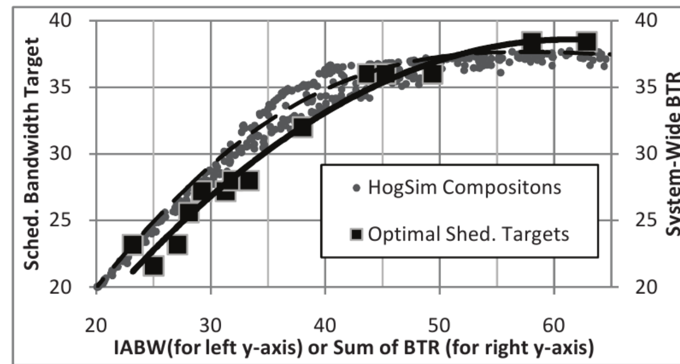

Figura 7: Aproximación entre el IABW y el ancho de banda de referencia óptimo

Lo siguiente que hacen los autores es contrastar la mejora del rendimiento con respecto al planificador nativo del sistema operativo. Para ello se comparan distintas políticas de planificación, empleando como valor de referencia el tope del ancho de banda (*PBW*), el ancho de banda medio ideal (*IABW*) y el ancho de banda óptimo calculado anteriormente por el método de tanteo, haciendo uso de las cargas de trabajo piloto. La figura 8 muestra los resultados de esta comparación.

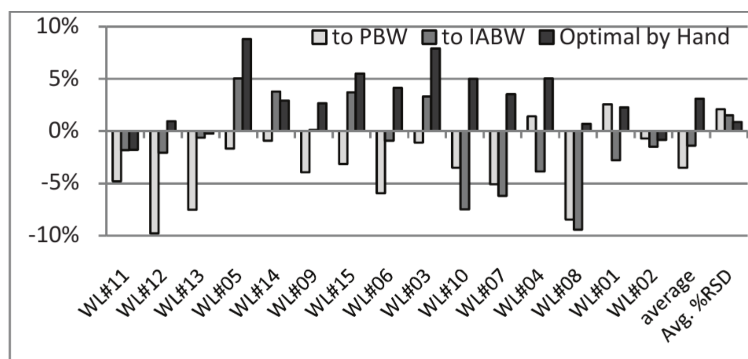

Figura 8: Mejora del rendimiento sobre el planificador nativo

Con el ajuste manual del ancho de banda de referencia óptimo, el rendimiento obtenido es de hasta un 8'8%, mientras que el rendimiento medio es de un 3'1%.

A continuación se utilizan los dos grupos de cargas de trabajo restantes para medir la mejora del rendimiento con respecto al planificador nativo del sistema operativo. Estas cargas de trabajo no tienen ningún valor de referencia óptimo calculado, en su lugar se utiliza la fórmula obtenida para la

nueva política de planificación a partir de los resultados de las cargas de trabajo piloto, para comprobar cómo de general es esta ecuación usada en otras cargas de trabajo. Al mismo tiempo, se compara con la política de planificación que tiene como referencia el tope del ancho de banda (*PBW*). En las figuras 9 y 10, puede compararse la mejora con respecto al planificador nativo con cargas de trabajo *SPEC* y *NPB*, respectivamente.

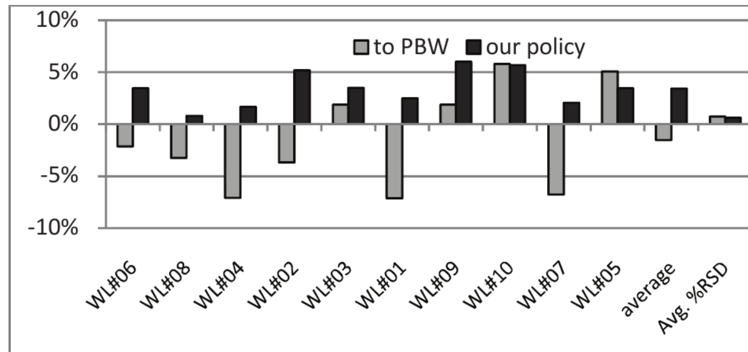

Figura 9: Mejora respecto al planificador nativo con las cargas de trabajo SPEC

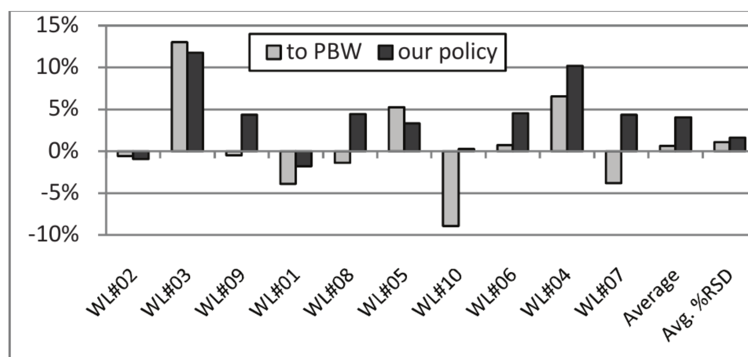

Figura 10: Mejora respecto al planificador nativo con las cargas de trabajo NPB

Estableciendo el ancho de banda de referencia del planificador en el tope del ancho de banda, la mayoría de las cargas de trabajo sufren una degradación en el rendimiento. La degradación media en las cargas de trabajo *SPEC* es del 1'4%, mientras que en el caso de las cargas de trabajo *NPB* la mejora es tan sólo del 1'1%.

Cuando se utiliza el ancho de banda de referencia de la nueva política de planificación, la mejora media es de un 3'4% en el caso de las cargas de trabajo *SPEC* y de un 4'9% en el caso de las *NPB*. Debido a la adaptabilidad del ancho de banda de referencia de planificación, la degradación del rendimiento más grave sufrida entre todas las cargas de trabajo es solamente del 1'8%.

Esta política puede adaptarse automáticamente a los cambios del ancho de banda de memoria utilizado por la carga de trabajo, por lo que funciona bien para cargas de trabajo con diferentes requisitos de ancho de banda.

3 Trabajo relacionado

3.1 Arquitectura de un procesador

Durante la realización del proyecto se ha trabajado con el procesador con nombre comercial *Intel Core 2 Duo processor T8100*. Este procesador de doble núcleo pertenece a la línea de procesadores de 45 nm de Intel, con el identificador CPUID 23 y de nombre *Penryn*, que forma parte de la microarquitectura *Core*.

3.1.1 Introducción a la microarquitectura *Core*

La microarquitectura *Core* fue presentada por Intel en el año 2006, como sucesora de la microarquitectura *P6*. Esta microarquitectura surge debido a las limitaciones y cuellos de botella que presentaba la microarquitectura *Netburst*, en la que estaban basados los procesadores *Pentium 4/D*, y que tuvo que ser abandonada por Intel. La evolución de las distintas microarquitecturas de Intel basadas en la arquitectura *IA-32* queda reflejada en la figura 11.

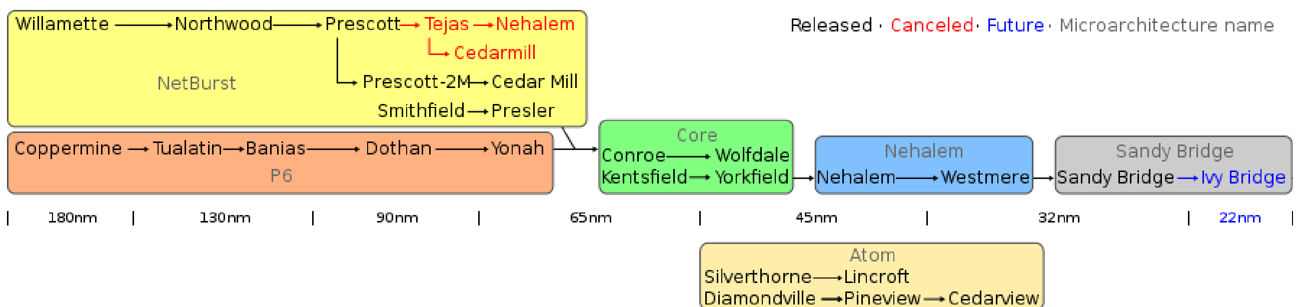

Figura 11: Evolución de la arquitectura IA-32

La microarquitectura *Core* se caracteriza por haber sido diseñada optimizada para multiprocesador. A diferencia de *Netburst*, se emplean frecuencias más bajas y se hace una utilización más eficiente de los ciclos de reloj disponibles. Esto hace que se reduzca el consumo energético al mismo tiempo que se incrementa la capacidad de proceso. Sin embargo, los procesadores basados en la microarquitectura *Core* no disponen de la tecnología *Hyper-Threading* presente en los procesadores *Pentium 4*. Esto es debido a que la microarquitectura *Core* es descendiente de la microarquitectura *P6* utilizada por los *Pentium Pro*, *Pentium II*, *Pentium III* y *Pentium M*.

3.1.2 La jerarquía de memoria

En la microarquitectura *Core* se dispone de una memoria caché de primer nivel (L1) para instrucciones y otra para datos por cada núcleo del procesador. En el caso de las versiones con dos núcleos, hay una única memoria caché de segundo nivel (L2) que es compartida por ambos, como

puede verse en la figura 12.

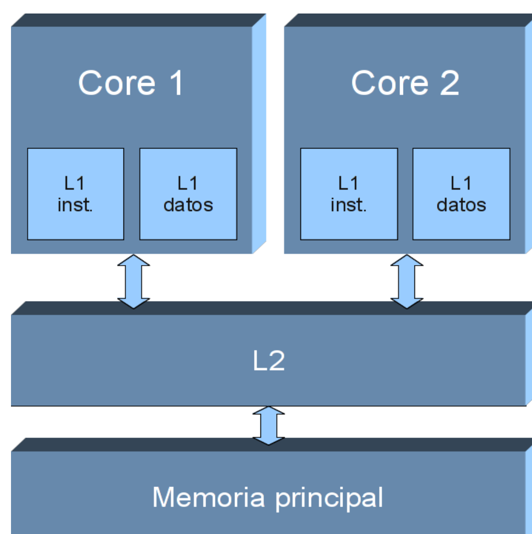

Figura 12: Jerarquía de memoria en la microarquitectura Core

Las características de la memoria caché del *T8100* aparecen especificadas en la figura 13.

Tipo de memoria caché	Tamaño	Correspondencia	Tamaño de línea	Política de reemplazo
L1 instrucciones	2 x 32 KBytes	Asociativa por conjuntos de 8 vías	64 bytes	-
L1 datos	2 x 32 KBytes	Asociativa por conjuntos de 8 vías	64 bytes	*Write back*
L2	3072 KBytes	Asociativa por conjuntos de 12 vías	64 bytes	-

Figura 13: Características de la memoria caché del T8100

Por otro lado, el bus frontal (*FSB*) que interconecta la caché L2 con la memoria principal funciona a 800 Mhz. Es importante comentar también que la cantidad total de memoria principal instalada en el sistema con el que se ha trabajado durante la realización del proyecto es de 3 GBytes.

Para realizar la compartición de la caché L2, se emplea la tecnología de Intel *Advanced Smart Cache*, que permite a cada núcleo utilizar dinámicamente el 100% de la caché L2 disponible. Cuando un núcleo tiene unos requisitos mínimos de utilización de caché, el otro núcleo puede incrementar su porcentaje de uso de caché L2, reduciendo los fallos de caché e incrementando el rendimiento. Esto también permite la obtención de datos desde la caché a unas tasas de

transferencia mayores. Por otro lado, cuando los dos núcleos requieren acceso a datos comunes, éstos son almacenados una sola vez de forma que ambos son capaces de acceder a los mismos. Todo esto permite obtener una mayor tasa de aciertos en la caché, una reducción del tráfico en el bus y una menor latencia de acceso a datos. En la figura 14 puede verse un esquema básico del funcionamiento de este sistema.

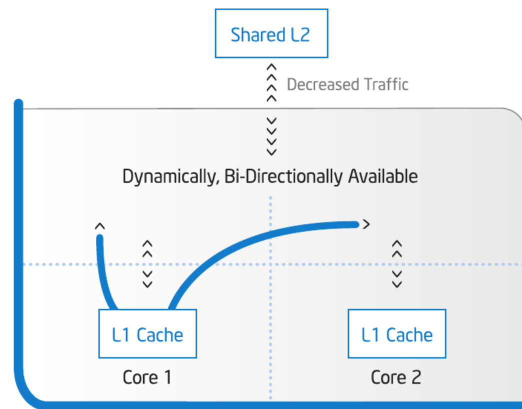

Figura 14: Compartición de la caché L2

3.1.3 La unidad de monitorización del rendimiento

La unidad de monitorización del rendimiento (PMU) es una infraestructura proporcionada por el hardware que permite la acumulación y obtención de datos de rendimiento del sistema a través de eventos que generan señales cuando se producen. Hay multitud de eventos hardware soportados, pero sólo unos pocos pueden ser contabilizados de forma simultánea, dependiendo de las características de la PMU.

La información de la PMU puede obtenerse mediante la instrucción *CPUID*, pasando el valor 0x0A en el registro *EAX* (CPUID.0AH). Esta información será devuelta en los registros *EAX*, *EBX* y *EDX*. La monitorización de rendimiento arquitectural estará soportada si el identificador de versión es mayor que 0 (bits 7:0 del registro *EAX*). El procesador *Intel Core 2 Duo processor T7700* y los procesadores posteriores basados en la microarquitectura *Core* (entre los que se encuentra el *T8100*), soportan la versión 2.

Existen dos tipos de eventos, los eventos arquitecturales y los eventos no arquitecturales.

- Los eventos arquitecturales son aquellos que se comportan de forma consistente entre microarquitecturas. Los bits a cero en el registro *EBX* tras ejecutar la instrucción CPUID.0AH indican que el evento está disponible. La lista de eventos arquitecturales y su correspondencia con el registro *EBX* queda reflejada en la figura 15.

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

Figura 15: Lista de eventos arquitecturales

- Los eventos no arquitecturales son específicos de cada microarquitectura. En el caso de la microarquitectura *Core*, hay unos 400, especificados en el manual *Intel(R) 64 and IA-32 architectures software developer's manual, volume 3B* [5].

La unidad de monitorización de rendimiento está formada por colecciones de registros MSR (*Model Specific Register*). Los registros MSR pueden ser registros contadores y registros de control de los contadores. Se puede acceder a estos registros mediante las instrucciones *RDMSR* y *WRMSR* si se ejecutan en el nivel de privilegio 0 del procesador. Ciertos registros contadores también pueden ser leídos mediante la instrucción *RDPMC*, desde cualquier nivel de privilegio.

Existen dos tipos de registros contadores, los de función fija y los de propósito general. Los eventos no arquitecturales sólo pueden ser recogidos por los contadores de propósito general. Los eventos arquitecturales pueden ser recogidos tanto por contadores de propósito general como por contadores de función fija.

La información de los registros contadores que hay por cada núcleo del procesador se puede obtener mediante la instrucción CPUID.0AH. En el registro *EDX*, se obtiene el número de contadores de función fija (bits 4 a 0) y su tamaño en bits (bits 12 a 5). En el registro *EAX*, se obtiene el número de contadores de propósito general (bits 15 a 8) y su tamaño en bits (bits 23 a 16). En la microarquitectura *Core*, hay 3 contadores de función fija y 2 de propósito general por cada núcleo del procesador. El tamaño de cada contador es de 40 bits en ambos casos.

Los registros contadores de función fija son llamados *IA32_FIXED_CTRx*. Cada uno de ellos se dedica a contar un evento arquitectural predefinido, que puede verse en la figura 16.

Event Name	Fixed-Function PMC	PMC Address
INST_RETIRED.ANY	MSR_PERF_FIXED_CTR0/IA32_FIXED_CTR0	309H
CPU_CLK_UNHALTED.CORE	MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH
CPU_CLK_UNHALTED.REF	MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH

Figura 16: Asociación de los contadores de función fija con eventos arquitecturales

Los registros contadores de función fija comparten un único registro de control, llamado *IA32_FIXED_CTR_CTRL*. Su estructura puede verse en la figura 17. Este registro permite habilitar la cuenta en el nivel del sistema operativo, en el nivel de usuario, en todos los niveles o deshabilitar la cuenta para cada uno de los registros contadores. También permite habilitar y deshabilitar el envío de interrupciones cuando se produce un *overflow* en un contador.

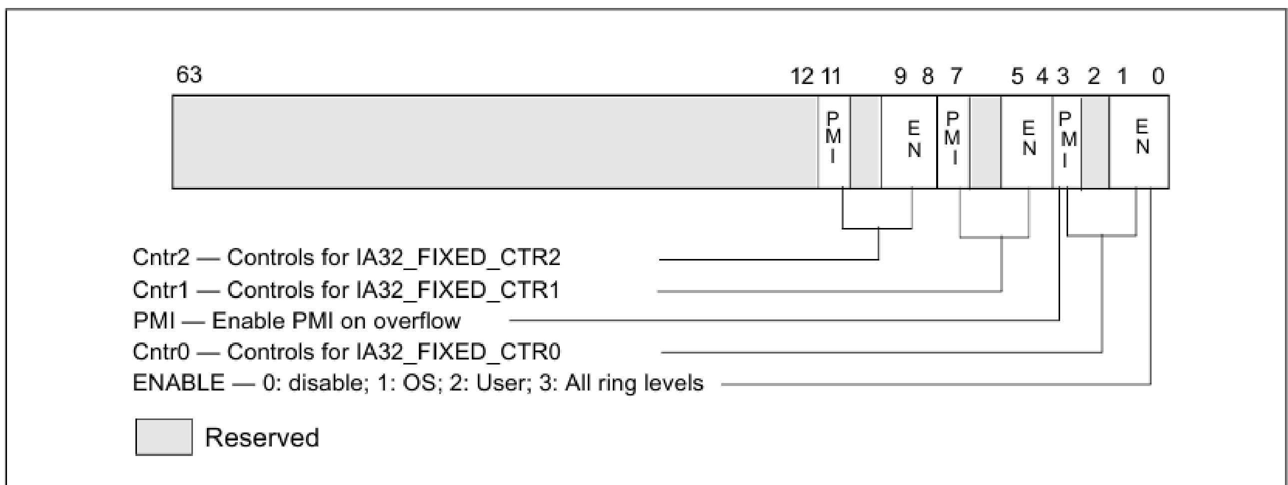

Figura 17: Estructura del registro de control IA32_FIXED_CTR_CTRL

Los registros contadores de propósito general son llamados *IA32_PMCx* y los de control *IA32_PERFEVTSELx*. Cada registro contador de propósito general tiene asociado un registro de control. Los registros de control de propósito general son más complejos que el registro de control de función fija, debido a que necesitan información detallada del evento que se ha de programar. En la figura 18 puede verse la estructura del registro de control de propósito general. Event select y Unit mask identifican al evento. Los valores para los eventos arquitecturales se pueden consultar en la lista de eventos arquitecturales de la figura 15. Para los eventos no arquitecturales, se debe consultar el manual *Intel(R) 64 and IA-32 architectures software developer's manual, volume 3B* [5].

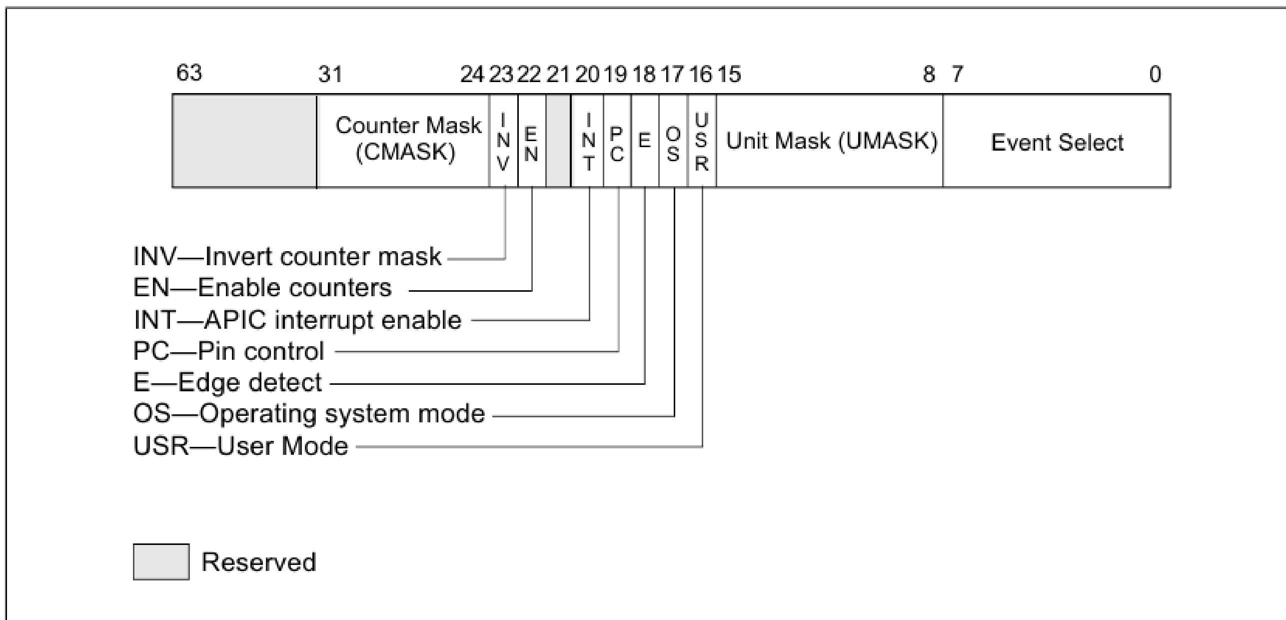

Figura 18: Estructura del registro de control IA32_PERFEVTSELx

También existen registros de control globales, que simplifican las operaciones más frecuentes en la programación de eventos. Esto es posible gracias a que con una sola instrucción sobre estos registros pueden verse involucrados todos los registros contadores o cualquier combinación de ellos, independientemente de si son contadores de función fija o de propósito general. Los registros de control globales son 3:

- *IA32_PERF_GLOBAL_CTRL* habilita y deshabilita la cuenta de eventos mediante una única instrucción *WRMSR*.
- *IA32_PERF_GLOBAL_STATUS* permite consultar los estados de *overflow* mediante una única instrucción *RDMSR*.
- *IA32_PERF_GLOBAL_OVF_CTRL* permite limpiar los estados de *overflow* mediante una única instrucción *WRMSR*.

3.2 Los contadores de rendimiento en el sistema operativo

Los contadores de rendimiento aparecen por primera vez de forma oficial en la versión de Linux 2.6.31. Previamente había otros proyectos que permitían obtener la misma funcionalidad. Estos proyectos consistían por un lado en un parche que al aplicarlo al núcleo ofrecía una interfaz de llamadas al sistema y por otro lado en una biblioteca de usuario que implementaba la infraestructura de los eventos para las distintas arquitecturas, permitiendo trabajar de forma más sencilla. Entre estos proyectos destacan *perfctr* y *perfmon2*, proyectos con un varios años de desarrollo y experiencia.

perfmon2 implementaba 12 llamadas al sistema que permitían trabajar con un sistema de contadores de rendimiento. La idea era que fuera introducido en la rama principal del núcleo, sin embargo nunca fue aceptado. En un intento por cambiar esto, el autor de *perfmon2*, Stéphane Eranian, propuso un nuevo parche que simplificaba la interfaz de llamadas al sistema, reduciendo el número de llamadas a 5, y pasando a llamarse *perfmon3*. Esta versión fue tenida en cuenta por varios desarrolladores del kernel, que comenzaron a desarrollar por su cuenta una nueva implementación que trata de alcanzar el mismo resultado final, pero basada en un concepto distinto en el que únicamente se añade una nueva llamada al sistema a la lista de llamadas existente.

3.2.1 La llamada al sistema *sys_perf_event_open()*

El subsistema de contadores de rendimiento de Linux está concebido de forma que hay una única llamada al sistema para crear contadores. Para poder trabajar posteriormente con estos contadores, hay que hacer uso de las operaciones típicas del sistema de ficheros virtual de Linux (VFS), como por ejemplo operaciones de lectura o de bloqueo.

El núcleo proporciona contadores virtuales de 64 bits, independientemente del tamaño de los contadores hardware subyacentes (cabe recordar que en el caso del *T8100,* son de 40 bits). Esto implica que debe encargarse del control de *overflow* cuando el tamaño del registro contador es más pequeño que el del contador virtual.

La idea de hacer uso de contadores virtuales, es que los registros jamás queden expuestos al espacio de usuario. Para lograr este objetivo, se prescinde de una biblioteca de usuario, de modo que toda la infraestructura de los eventos para las diferentes arquitecturas se encuentra implementada en el propio núcleo. La figura 19 muestra la diferencia entre esta implementación y otras versiones.

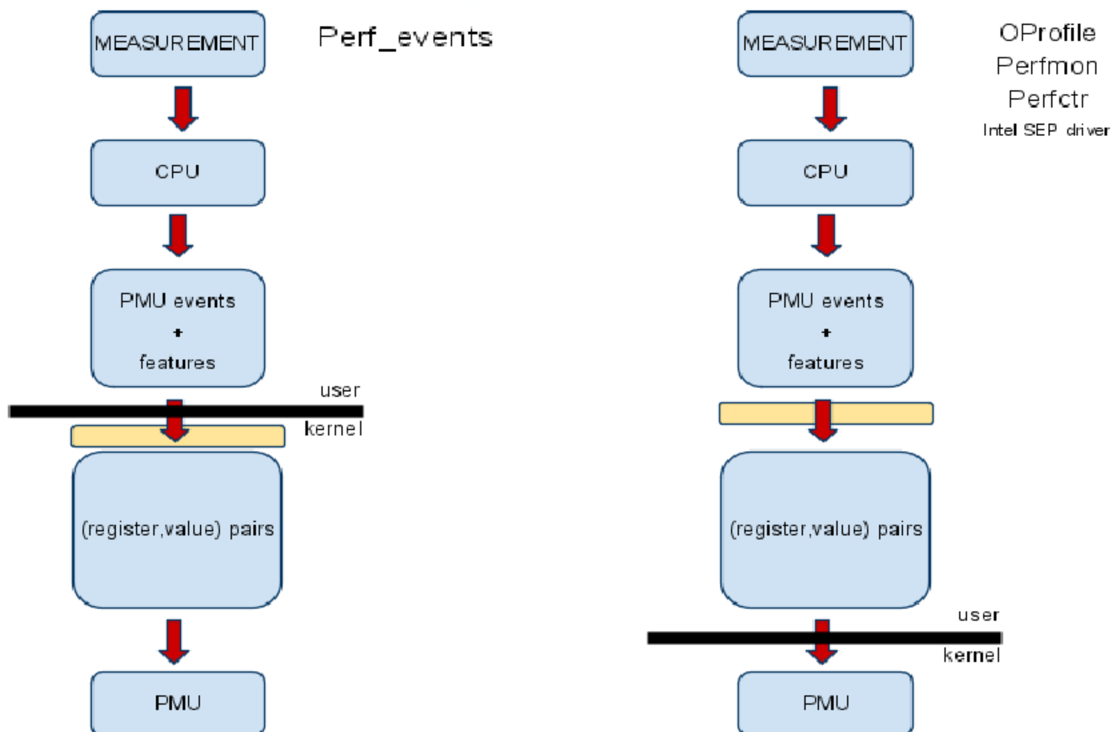

Figura 19: API orientada a eventos contra API orientada a registros

Para poder acceder a los contadores de rendimiento en Linux, se hace uso de descriptores de fichero especiales, de modo que cada descriptor de fichero utilizado permite acceder a un contador virtual. Estos descriptores de fichero especiales se abren mediante la llamada al sistema *sys_perf_event_open()*. A partir de ese momento el evento programado comienza a ser contabilizado. De este modo, cada evento que se vaya a programar, requiere realizar una llamada al sistema *sys_perf_event_open()*.

En caso de que el número de eventos hardware que se programen simultáneamente supere al número de registros de que dispone la unidad de monitorización del rendimiento (PMU) de la CPU, el núcleo multiplexará los contadores, de modo que todos los eventos puedan ser contabilizados. Esto se consigue realizando una planificación de tipo *round-robin* por grupos de eventos, de forma que los eventos agrupados son monitorizados al mismo tiempo (los eventos sin agrupar pueden verse también como grupos de eventos de un sólo miembro). De este modo, el número de eventos hardware por grupo no podrá superar al número de registros de que dispone la PMU del hardware.

En la figura 20 puede verse un ejemplo de planificación circular con 4 grupos de eventos. En cada tic de reloj, el siguiente grupo de la cola pasa a ocupar los registros contadores.

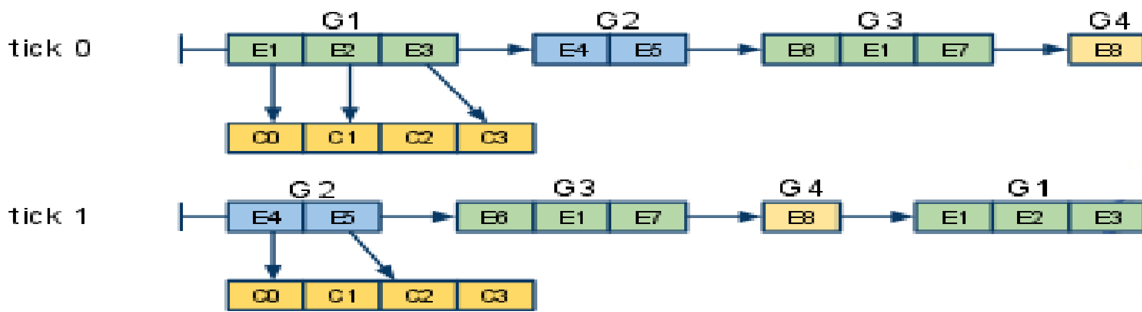

Figura 20: Planificación round-robin de grupos de eventos.

Cuando se utiliza multiplexación, los datos recogidos deben ser posteriormente escalados, ya que un porcentaje de los valores se pierde.

En la práctica, el factor de escala puede variar entre los distintos grupos de eventos que están siendo monitorizados. Esto dependerá en gran medida de si se permite o no contabilizar a varios grupos simultáneamente cuando se dispone de registros disponibles para todos ellos. En las figuras 21 y 22 puede verse un ejemplo de cada tipo.

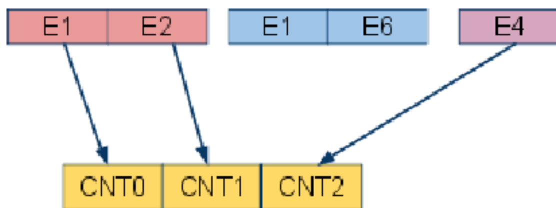

Figura 21: Diferentes grupos de eventos pueden ser contabilizados simultáneamente (exclusive=0)

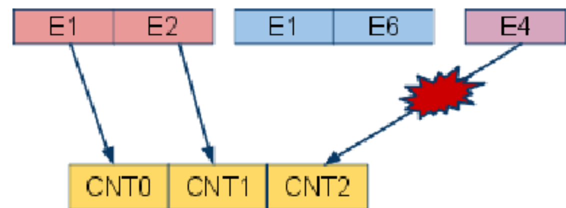

Figura 22: No está permitida la monitorización simultánea de grupos de eventos (exclusive=1)

Así, la forma adecuada de calcular el factor de escala es a partir de los valores de multiplexación del evento *tiempo total* y *tiempo activo*:

$$Factor\,de\,escala = \frac{t_{total}}{t_{activo}}$$

Donde t_{total} es el tiempo que la tarea o grupo de tareas a las que está asociado el evento ha sido monitorizado y t_{activo} es el tiempo durante el cual el evento o grupo de eventos ha sido contabilizado. El valor final obtenido para cada evento, deberá ser posteriormente multiplicado por su factor de escala correspondiente. Más adelante se verá cómo realizar la lectura del *tiempo total* y el *tiempo activo*.

La interfaz de la llamada al sistema es la siguiente:

```
int sys_perf_event_open(struct perf_event_attr *attr_uptr,
        pid_t pid, int cpu, int group_fd, unsigned long flags)
```

La llamada al sistema devuelve el nuevo descriptor de fichero. El descriptor de fichero puede ser utilizado con las llamadas al sistema habituales:

- *read()* permite leer el contador.
- *close()* cierra el descriptor.
- *fcntl()* puede ser utilizada para establecer el modo de bloqueo.
- *poll()* espera un evento en un descriptor de fichero.

El parámetro de tipo *perf_event_attr* define cómo se comportará el evento. Es una estructura compleja, con un considerable número de campos.

```
struct perf_event_attr {
        __u32                   type;
        __u32                   size;
        __u64                   config;
        union {
                __u64           sample_period;
                __u64           sample_freq;
        };
        __u64                   sample_type;
        __u64                   read_format;

        __u64                   disabled       :  1,
                                inherit        :  1,
                                pinned         :  1,
                                exclusive      :  1,
                                exclude_user   :  1,
                                exclude_kernel :  1,
                                exclude_hv     :  1,
                                exclude_idle   :  1,
                                mmap           :  1,
                                comm           :  1,
                                freq           :  1,
                                inherit_stat   :  1,
                                enable_on_exec :  1,
                                task           :  1,
                                watermark      :  1,
                                precise_ip     :  2,
                                mmap_data      :  1,
                                sample_id_all  :  1,

                                __reserved_1   : 45;

        union {
                __u32           wakeup_events;
                __u32           wakeup_watermark;
        };

        __u32                   bp_type;
        union {
                __u64           bp_addr;
                __u64           config1;
        };
        union {
                __u64           bp_len;
                __u64           config2;
        };
};
```

Los tres campos básicos de la estructura *perf_event_attr* que hay que definir para programar un evento son *type*, *size* y *config*.

- *type* indica el tipo de evento que se ha de monitorizar. Los eventos pueden ser de varios

tipos, entre los cuales destacan:

- *PERF_TYPE_HARDWARE*. Son eventos hardware genéricos, que el núcleo se encarga de traducir internamente en eventos específicos de la arquitectura, como por ejemplo instrucciones de salto realizadas.
- *PERF_TYPE_SOFTWARE*. Son eventos software definidos por el núcleo, como fallos de página o cambios de contexto. Pueden ser empleados en arquitecturas que no dispongan de la unidad de monitorización del rendimiento hardware.
- *PERF_TYPE_HW_CACHE*. Son eventos hardware genéricos, pero dedicados exclusivamente a la caché, como fallos de lectura en la caché de datos de primer nivel.
- *PERF_TYPE_RAW*. Son eventos específicos de la microarquitectura que hay que pasar mediante su codificación en hexadecimal, fuertemente relacionada con la codificación de bits del registro de control. Para ayudar en la codificación de eventos *RAW*, la biblioteca *libpfm4* [14] y el manual de Intel [5] son útiles.

• *size* indica el tamaño de la estructura *attr*, para mejorar la compatibilidad entre versiones del núcleo. Hay que establecerla con *sizeof(struct perf_event_attr)*.

• *config* permite especificar el evento deseado, con ayuda del campo *type*. Si la CPU no es capaz de contabilizar el evento seleccionado, entonces la llamada al sistema devolverá -*EINVAL*.

Otros campos de la estructura *perf_event_attr* que resultan de interés son r*ead_format*, *disabled*, *inherit* y *exclusive*.

- *read_format* permite definir el formato de los datos que serán leídos al realizar una operación *read()* sobre el descriptor de fichero del evento. Estos datos extra se añadirán a continuación del valor del contador. Las opciones disponibles son:
  - *PERF_FORMAT_TOTAL_TIME_RUNNING* y *PERF_FORMAT_TOTAL_TIME_ENABLED* , que permiten obtener el *tiempo total* y el *tiempo activo* de un evento, para poder escalar los valores posteriormente, añadiendo un campo de 64 bits cada uno.
  - *PERF_FORMAT_ID* añade un identificador del grupo de eventos de 64 bits.
  - *PERF_FORMAT_GROUP* permite leer todos los valores de un grupo de eventos con una sola llamada *read()*.
- *disabled* permite programar un evento inicialmente desactivado. Para habilitarlo posteriormente, habrá que hacer uso de la llamada *ioctl()*:

```
ioctl(fd, PERF_EVENT_IOC_ENABLE);
```

- *inherit* indica si para contabilizar se ha tener en cuenta también a los procesos hijos de la tarea actual que se creen con la llamada *fork()*, a partir del instante en que se programa el evento.
- *exclusive* define si se impedirá contabilizar diferentes grupos de eventos al mismo tiempo, cuando haya suficientes registros hardware disponibles para hacerlo. Esto hace que en cada tic de reloj haya un sólo grupo que esté siendo contabilizado. El efecto del uso de este campo puede verse en las figuras 21 y 22 anteriormente citadas.

Los otros cuatro parámetros de la llamada al sistema *sys_perf_event_open()* son *pid*, *cpu*, *group_fd* y *flags*,, aunque el parámetro *flags* actualmente no se utiliza.

El parámetro *pid* permite que el contador sea específico de una tarea:

- Si es cero, el contador se asocia a la tarea actual.
- Si es mayor que cero, el contador se asocia a la tarea específica, siempre y cuando la tarea actual tenga los privilegios necesarios para hacerlo.
- Si es menor que cero, el contador es asignado a todas las tareas, son los llamados contadores por CPU.

El parámetro *cpu* permite que un contador sea específico de una CPU:

- Si es mayor o igual que cero, el contador se restringe a una CPU específica.
- Si es igual a -1, el contador cuenta en todas las CPUs. No es posible combinar esta opción con un valor de *pid* menor que cero, es decir, hacer que el contador sea asignado a todas las CPUs y a todas las tareas al mismo tiempo.

El parámetro *group_fd* permite monitorizar los eventos por grupos. Para crear un grupo de eventos, hay que programar un primer evento con *group_fd* igual a -1, que será el líder del grupo, y los eventos que se programen posteriormente, han de tener por *group_fd* el descriptor de fichero del líder del grupo. En la figura 23 puede verse un ejemplo simplificado de agrupación con dos eventos, haciendo uso de este parámetro.

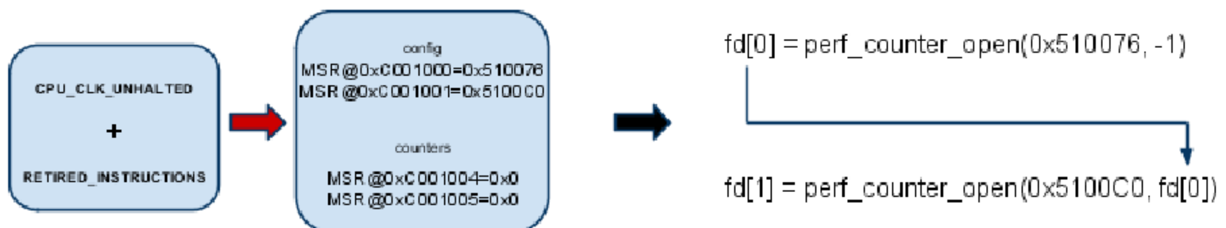

Figura 23: Agrupación de eventos

3.3 Benchmarks

Para la realización de las experiencias, es preciso hacer uso de pruebas de rendimiento o *benchmarks*. Los *benchmarks* son aplicaciones que tienen un comportamiento predecible cuando se utilizan datos de entrada fijos en todas las ejecuciones. Esto permite medir y comparar los resultados obtenidos bajo distintos escenarios, y realizar análisis o ajustes en el sistema. Un *benchmark* puede abarcar todo el sistema en general, aunque habitualmente están divididos por categorías, de modo que cada uno tiene como objetivo poner a prueba una parte concreta del mismo. En este proyecto, se ha utilizado el paquete de *benchmarks SPEC CPU2006*.

3.3.1 SPEC CPU2006

Standard Performance Evaluation Corporation (SPEC) es una organización sin ánimo de lucro formada para establecer, mantener y apoyar un conjunto estándar de *benchmarks* de relevancia que puedan ser aplicados a la nueva generación de ordenadores de alto rendimiento [15]. Los conjuntos de *benchmarks* que desarrolla el SPEC, son ampliamente utilizados para evaluar el rendimiento de los sistemas informáticos. Estos resultados son posteriormente publicados en el sitio web del SPEC, de modo que puedan servir como valores de referencia para diferentes configuraciones hardware.

Dentro de estos grupos de *benchmarks*, *CPU2006* es de especial interés para el propósito de este proyecto. *CPU2006* ofrece un conjunto de pruebas estandarizadas desarrolladas a partir de aplicaciones de usuario reales, que permiten realizar pruebas intensivas de la CPU, haciendo hincapié en el procesador del sistema, el subsistema de memoria y el compilador.

CPU2006 está dividido a su vez en dos grupos de *benchmarks*: *CINT2006* para medir y comparar el rendimiento de la computación intensiva con enteros y *CFP2006* para medir y comparar el rendimiento de la computación intensiva en coma flotante.

Cada aplicación tiene un comportamiento distinto en cuanto a tiempo de finalización, tamaño de memoria virtual y residente [16], número de accesos a caché, consumo de ancho de banda del bus, etc. Algunas de estas peculiaridades pueden ser medidas gracias al uso de los contadores de rendimiento, que hacen posible la caracterización de dichas aplicaciones.

En total, *CPU2006* ofrece 31 *benchmarks*, de los cuales 12 son de enteros y 18 de coma flotante. A continuación se muestra una breve descripción [17] de cada uno de ellos.

Benchmarks de enteros (*CINT2006*):

- *400.perlbench* es una versión recortada de *Perl*. Además del intérprete de *Perl*, se utilizan varios módulos de terceros. La carga de trabajo principal consta de un programa de marcado de *spam*, un conversor de correo electrónico a HTML y una versión modificada de *specdiff*, la cual es parte de *CPU2006*.
- *401.bzip2* está basado en *bzip2*. La diferencia radica en que la versión del SPEC realiza todo el proceso de compresión y descompresión en memoria principal, evitando el uso del disco. De este modo se consigue aislar el trabajo en la CPU y el subsistema de memoria.
- *403.gcc* está basado en el compilador *GCC*. Este *benchmark* lanza un compilador que

genera código para el procesador de AMD *Opteron*.

- *429.mcf* proviene de *MCF*, un programa utilizado para la planificación de vehículos en el transporte público. El problema puede ser formulado como un problema de flujo a gran escala de coste mínimo, resuelto con un algoritmo de red simplex.
- *445.gobmk* está basado en *GNU Go*. El programa se dedica a jugar al antiguo juego de tablero chino *Go*, ejecutando un conjunto de órdenes para analizar las posiciones.
- *456.hmmer* trabaja con una técnica utilizada en biología computacional para buscar patrones en secuencias de ADN. La técnica es utilizada para analizar secuencias de proteínas, realizando búsquedas en una base de datos con descripciones estadísticas.
- *458.sjeng* está basado en *Sjeng*, programa que juega al ajedrez y varias de sus variantes. El programa intenta encontrar para cada jugador el mejor movimiento, explorando las distintas ramas del árbol desde una posición dada hasta una profundidad determinada, extendiendo las interesantes y descartando las dudosas o irrelevantes.
- *462.libquantum* utiliza la implementación de un algoritmo incluido en la biblioteca *libquantum* para factorizar números primos en tiempo polinómico. El problema de factorizar números primos es de gran utilidad en el análisis criptográfico, ya que hay algoritmos ampliamente utilizados como RSA que están basados en él.
- *464.h264ref* está basado en la implementación de referencia *H264/AVC*, un estándar de compresión de vídeo muy extendido. El programa codifica vídeo utilizando dos perfiles. En el primer perfil, se busca un ratio de compresión que permita una codificación rápida, útil para aplicaciones de tiempo real como videoconferencia. El segundo perfil busca la mejor compresión, útil para aplicaciones donde no puede haber pérdida de datos, como reproductores DVD.
- *471.omnetpp* consiste en la simulación de una red Ethernet grande, basándose en el simulador *OMNet++*. El modelo empleado simula una red Ethernet de un campus universitario que genera tráfico entre los dispositivos de red.
- *473.astar* está basado en una biblioteca de búsqueda de caminos que se utiliza en la inteligencia artificial de videojuegos, implementando en total tres algoritmos distintos. El *benchmark* trata de simular la búsqueda de caminos en un mapa y medir su longitud.
- *483.xalancbmk* es una versión modificada de *Xalan-C++*, un procesador XSLT para transformar documentos XML en HTML, texto u otros tipos de documentos XML. El procesador XSLT recibe una hoja de estilo XSL que especifica la conversión necesaria de un árbol de nodos (fichero XML de entrada) a otro (fichero de salida resultante).
- *998.specrand* utiliza un algoritmo de generación números aleatorios para generar una secuencia de números pseudoaleatorios que empiezan con una semilla conocida. *specrand* no es una prueba de tiempo, en lugar de ello se utiliza como un rápido indicador de problemas, ya que varios *benchmarks* de *CPU2006* emplean este código como su generador de números pseudoaleatorios.

Benchmarks de coma flotante (*CFP2006*):

- *410.bwaves* simula numéricamente ondas de choque en un flujo viscoso tridimensional. El algoritmo implementado resuelve las ecuaciones de Navier-Stokes utilizando el método *Bi-CGstab*, que resuelve sistemas de ecuaciones lineales no simétricas iterativamente.

- *416.gamess* está basado en *GAMESS*, un paquete de computación química cuántica. Este *benchmark* utiliza el método directo SCF para realizar la computación de: 1) la molécula citosina, 2) agua y Cu2+ y 3) el ión triazolium.
- *433.milc* emplea *MILC*, un programa para realizar simulaciones de la teoría de campos en el retículo de cuatro dimensiones. El *benchmark* genera un campo gauge que se utiliza con aplicaciones de la teoría de campos en el retículo que involucran a quarks dinámicos.
- *434.zeusmp* está basado en *ZEUS-MP*, que emplea dinámica de fluidos para la simulación de fenómenos astrofísicos. El programa resuelve ecuaciones de hidrodinámica y magneto-hidrodinámica, incluyendo campos gravitatorios.
- *435.gromacs* proviene de *GROMACS*, un paquete que realiza la simulación de las ecuaciones newtonianas del movimiento para sistemas con cientos de millones de partículas. La versión del benchmark realiza la simulación de la proteína Lisozima en una solución de agua e iones.
- *436.cactusADM* es una combinación del entorno de resolución de problemas *Cactus* y del núcleo computacional *BenchADM*. *CactusADM* resuelve las ecuaciones de Einstein, las cuales son un conjunto de diez ecuaciones diferenciales parciales no lineares que relacionan la presencia de materia con la curvatura del espacio-tiempo.
- *437.leslie3d* proviene de *LESlie3d*, un programa de dinámica de fluidos computacional utilizado para investigar una amplia gama de fenómenos de turbulencia, como la mezcla, la combustión y la acústica. Este *benchmark* resuelve un problema utilizando la capa de mezcla temporal, un tipo de flujo que se produce en las cámaras de combustión que utilizan inyección de combustible.
- *444.namd* proviene de *NAMD*, un programa para simular grandes sistemas biomoleculares. La mayor parte del tiempo de ejecución se gasta en el cálculo de interacciones atómicas. El *benchmark* estándar simula una apolipoproteína A-1 de 92224 átomos.
- *447.dealII* utiliza *deal.II*, una biblioteca dirigida a la solución de ecuaciones diferenciales parciales mediante el método de los elementos finitos. El caso de prueba de este *benchmark* resuelve una ecuación de Helmholtz en 3d con coeficientes no constantes.
- *450.soplex* está basado en *SoPlex*, un módulo para resolver modelos de programación lineal utilizando el método *Simplex*. En este benchmark, se trabaja con matrices dispersas, empleando la factorización LU y rutinas de resolución para los sistemas de ecuaciones triangulares resultantes.
- *453.povray* emplea *POV-Ray*, un trazador de rayos que calcula la imagen de una escena simulando la forma en que los rayos de luz viajan en el mundo real. Las intersecciones de los rayos con objetos geométricos se computan resolviendo directamente complejas ecuaciones matemáticas o con algoritmos de aproximación numérica.
- *454.calculix* está basado en *CalculiX*, un software de resolución de problemas por el método de los elementos finitos, con aplicaciones en la mecánica de estructuras tridimensionales lineares y no lineares. El ejemplo de referencia de este *benchmark* describe la deformación de un disco compresor debido a la fuerza centrífuga.
- *459.GemsFDTD* está basado en un subconjunto de *General ElectroMagnetic Solvers* (GEMS). Este *benchmark* resuelve las ecuaciones de Maxwell en 3D utilizando el método de diferencias finitas en el dominio del tiempo.
- *465.tonto* emplea *Tonto*, un paquete de química cuántica que dedica una parte importante a

la evaluación de integrales entre productos de funciones de base gaussiana. El *benchmark* trabaja en el campo de la cristalografía de rayos X, calculando una función de onda molecular por el método Hartree-Fock.

- *470.lbm* implementa el método de Lattice Boltzmann para simular fluidos incompresibles, una parte computacionalmente importante de un programa para simular la formación y el movimiento de burbujas de gas en espumas metálicas.
- *481.wrf* está basado en el modelo de investigación y predicción meteorológica *WRF*. *WRF* consta de un sistema de mesoescala numérica diseñado para servir tanto a las necesidades de predicción meteorológica como a las de investigación atmosférica.
- *482.sphinx3* está basado en *Sphinx-3*, un sistema de reconocimiento de voz. Este *benchmark* lee todas las entradas en el proceso de inicialización y las procesa repetidamente con diferentes ajustes.
- *999.specrand* es exactamente igual que *998.specrand*, pero trabaja con números en coma flotante, en lugar de enteros.

4 Experimentos y datos

4.1 Desarrollo de una herramienta para la toma de datos

Para poder realizar la caracterización de las aplicaciones, es necesario disponer de alguna herramienta que permita la recogida y el tratamiento de los datos de los contadores de rendimiento. En este caso, la herramienta implementada obtendrá los datos derivados de la ejecución de los *benchmarks*.

4.1.1 Diseño

El diseño de la herramienta de caracterización está basado en un modelo que utiliza pequeñas aplicaciones en espacio de usuario, las cuales trabajan con ficheros de datos. Estas aplicaciones pueden ser utilizadas de forma independiente, pasando de forma adecuada los parámetros a través de la línea de órdenes. Esto ha sido de gran utilidad durante la realización del proyecto para poder ensayar con los datos generados anteriormente sin tener que volver a ejecutar todos los pasos. Sin embargo, por simplicidad y automatización del proceso, fue necesario escribir un *guión* (también conocido en inglés como *script*) en *Bash*, que se encarga de realizar todos los pasos de forma ordenada, mediante el lanzamiento parametrizado del resto de aplicaciones. El esquema puede verse en la figura 24.

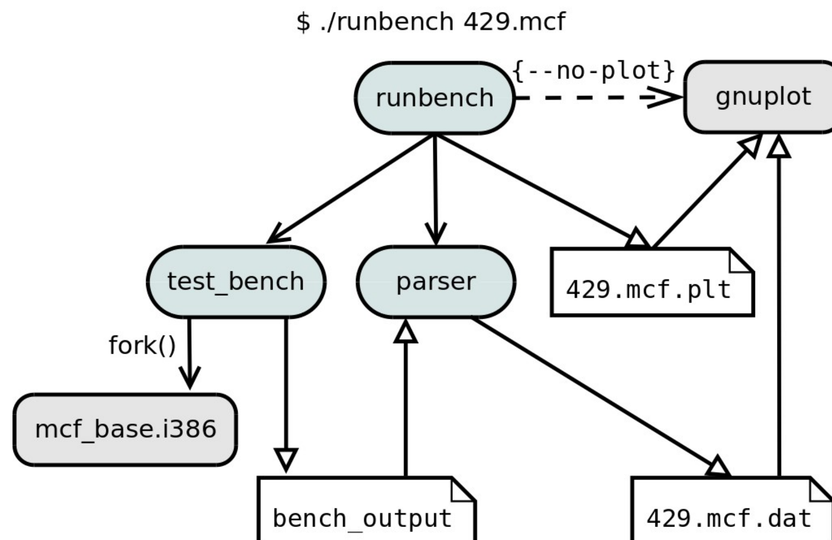

Figura 24: Esquema de ejecución de un benchmark

4.1.2 Lanzamiento de las aplicaciones

runbench es un guión escrito en *Bash* que se encarga de coordinar el lanzamiento del resto de aplicaciones con los parámetros adecuados.

Su utilización a través de la línea de órdenes es sencilla, requiriendo únicamente el nombre del *benchmark* que se debe lanzar. Opcionalmente, es posible indicar mediante la opción *--no-plot* que no se desea mostrar la gráfica con *gnuplot* al finalizar la ejecución, lo cual es útil cuando se quiere programar la ejecución de varios *benchmarks* de larga duración para su análisis posterior.

```
$ ./runbench [--no-plot] nombre_benchmark
```

En primer lugar, el guión debe comprobar si el *benchmark* indicado se encuentra en su lista de *benchmarks* conocidos, para asignarle los parámetros que necesita para iniciar su ejecución. La configuración de los parámetros de entrada de los *benchmarks* está basada en la información de la página del proyecto *Mult2Sim* [17].

En caso afirmativo, se lanza la aplicación *test_bench*, que recibe el nombre del fichero ejecutable (en este caso el *benchmark*) y la lista de argumentos del mismo. Esta aplicación escribe en un fichero binario los datos de la ejecución del *benchmark*.

Una vez finalizada la ejecución de *test_bench*, se lanza la aplicación *parser*. Esta aplicación recibe el nombre del fichero binario de datos anterior y transforma los datos en texto, en un formato por columnas que puede ser interpretado por *gnuplot*.

A continuación se genera un fichero de configuración de *gnuplot* para el *benchmark* anteriormente ejecutado y finalmente se lanza el programa *gnuplot*, para que muestre la gráfica con los resultados, siempre y cuando no se haya seleccionado la opción *--no-plot*.

4.1.3 Recogida de datos de los benchmarks

test_bench es la aplicación de mayor relevancia. Tiene como misión el lanzamiento de un *benchmark* y la recogida de los datos proporcionados por los contadores de rendimiento a lo largo de toda su ejecución. Para ello, se hace uso del sistema de contadores de rendimiento que ofrece Linux.

Definición del evento

En primer lugar hay que definir el evento que se pretende contabilizar. Para que sea posible caracterizar las aplicaciones, debe escogerse un evento que permita obtener el patrón de accesos a memoria en función del tiempo de referencia. Debido a que todos los accesos a memoria que se realizan han de pasar por el primer nivel de caché, se ha optado por el evento *L1D_ALL_REF* que aparece en el manual de Intel [5]. Este evento cuenta todas las referencias de cualquier tipo a la memoria caché de datos de primer nivel, incluyendo lecturas y escrituras.

L1D_ALL_REF es un evento no arquitectural que hay que codificar como RAW, lo cual tiene como desventaja una mayor complejidad, debido a la escasa abstracción que ofrece.

Según la lista de eventos no arquitecturales de la microarquitectura *Core* que aparece en el manual de Intel, este evento se codifica como Event Select = 43H y Unit Mask = 01H. Si se observa el formato del registro *IA32_PERFEVTSELx* (figura 18), se corresponde con los bits 7-0 y 15-8 del

registro, respectivamente. Así, el evento codificado quedaría como 0x0143. Sin embargo, faltan otros bits del registro que también hay que configurar, como la habilitación inicial (*EN = 1*) o si se desea contabilizar en modo núcleo (*OS = 1*) o usuario (*USR = 1*).

Otra opción más sencilla es emplear la biblioteca auxiliar *libpfm4* [14], escrita por el autor de *perfmon2* para funcionar con el sistema de contadores de rendimiento de Linux. Aunque esta biblioteca no ha sido estudiada en este proyecto para descubrir sus posibilidades, se han empleado dos de las utilidades de ayuda que ofrece.

Por un lado, mediante *showevtinfo*, es posible descubrir los eventos soportados disponibles:

```
$ ./libpfm-4.1.0/examples/showevtinfo | less
#-----------------------------
IDX      : 23068719
PMU name : core (Intel Core)
Name     : L1D_ALL_REF
Equiv    : None
Flags    : None
Desc     : All references to the L1 data cache
Code     : 0x143
Modif-00 : 0x00 : PMU : [k] : monitor at priv level 0 (boolean)
Modif-01 : 0x01 : PMU : [u] : monitor at priv level 1, 2, 3 (boolean)
Modif-02 : 0x02 : PMU : [e] : edge level (boolean)
Modif-03 : 0x03 : PMU : [i] : invert (boolean)
Modif-04 : 0x04 : PMU : [c] : counter-mask in range [0-255] (integer)
:
```

Por otro lado, una vez localizado el evento *L1D_ALL_REF*, se averigua su codificación, mediante la utilidad *evt2raw*:

```
$ ./libpfm-4.1.0/perf_examples/evt2raw -v L1D_ALL_REF
r530143 core::L1D_ALL_REF:e=0:i=0:c=0:u=1:k=1
```

La codificación RAW del evento *L1D_ALL_REF* que se ha obtenido es 0x530143. Se puede observar que por defecto aparecen activados los bits 22 (*Enable counters*), 20 (*APIC interrupt enable*), 17 (*Operating system mode*) y 16 (*User mode*) del registro *IA32_PERFEVTSELx*.

Funcionamiento de la aplicación

test_bench está escrita en *C*. Esta aplicación recibe como parámetros de entrada la ruta del ejecutable del *benchmark* junto a sus argumentos, para poder después ejecutarlo.

En primer lugar se inicializa la estructura *perf_event_attr* con la definición del evento.

```
...
#define L1D_ALL_REF 0x00530143
...
int main(int argc, char **argv)
```

```
{
...
  struct perf_event_attr attr_L1DAllRef =
  {
    .type = PERF_TYPE_RAW,
    .config = L1D_ALL_REF,
...
```

A continuación, la aplicación se divide en dos procesos mediante la llamada *fork()* y el hijo se suspende voluntariamente.

```
...
  childpid = fork();

  if (childpid == -1) { // Error
...
  } else if (childpid == 0) { // Hijo
    kill (getpid(), SIGSTOP);
...
```

El padre espera a que el hijo se suspenda mediante un pequeño bucle de espera activa, ya que posteriormente ha de enviarle una señal para despertarlo, y ésta podría perderse. El motivo de sincronizar al padre y al hijo es debido a que al ser dos procesos, el orden de ejecución lo establece el planificador y no es posible controlarlo ni hacer suposiciones de quién llegará antes a determinado punto del código.

```
...
  } else { // Padre
    while (waitpid(childpid, NULL, WUNTRACED) == 0)
      usleep(1000);
...
```

El proceso padre será el encargado de realizar la monitorización. Para que ésta pueda realizarse de forma más precisa, el padre intenta cambiar su tipo de proceso por uno de tiempo real. Esto sólo es posible si el proceso es lanzado con permisos de superusuario.

```
...
    if (geteuid() == 0)
      sched_setscheduler(0, SCHED_FIFO, &p);
...
```

Por otro lado, también prepara en modo escritura el fichero donde se van a volcar los datos de la monitorización.

```
...
    if ( (fd_output = open("/tmp/bench_output.data", O_WRONLY | O_CREAT |
    O_TRUNC, 0644)) == -1) {
      perror("No se pudo abrir el fichero de salida");
      kill (childpid, SIGTERM);
      exit(EXIT_FAILURE);
    }
...
```

A continuación el padre programa el evento anteriormente definido con la llamada al sistema *sys_perf_event_open()* y envía al hijo la señal *SIGCONT* para despertarlo.

```
...
    fd_L1DAllRef = sys_perf_event_open(&attr_L1DAllRef, childpid, -1, -1, 0);

    if (fd_L1DAllRef < 0) {
      perror("No se pudo obtener el descriptor del PMC");
      kill (childpid, SIGTERM);
      exit(EXIT_FAILURE);
    }

    kill (childpid, SIGCONT);
...
```

El hijo cambia su mapa de memoria por el del *benchmark* mediante la llamada *exec()*, haciendo uso de los parámetros de entrada que recibió la aplicación.

```
...
    if (execv(argv[1], &argv[1]) < 0) {
      perror("No se pudo ejecutar el programa");
      exit(EXIT_FAILURE);
    }
...
```

Para poder medir los valores intermedios, el padre entra en un bucle, en el cual obtiene tanto el valor del contador como el del tiempo de ejecución del proceso hijo, escribe los resultados en el fichero de salida y se suspende durante un tiempo previamente definido (por ejemplo, de 100 ms). Este bucle se repite hasta que el proceso hijo termina.

Debido a que la aplicación trabaja en espacio de usuario, no es posible obtener los datos del tiempo de ejecución del *benchmark* directamente de su estructura del núcleo *task_struct*. Por este motivo, hubo que plantear varias alternativas, hasta que se alcanzó la mejor forma posible.

En primer lugar se planteó la posibilidad de medir el tiempo mediante el uso de la función *gettimeofday()* definida en el fichero de cabecera *sys/time.h*. Esta función devuelve en una estructura de tipo *timeval* el número de segundos y microsegundos desde el 1 de enero de 1970. Con este valor, es posible calcular el tiempo que ha transcurrido entre una lectura y la siguiente.

```
...
            read(fd_L1DAllRef, &cont_L1DAllRef, sizeof(u64));
            gettimeofday(&ts, NULL);
...
```

Este método fue descartado, ya que no se obtiene el tiempo de ejecución, sino el tiempo transcurrido. Incluso aunque las experiencias se realicen con la mínima carga posible en el sistema y se trabaje con un procesador de doble núcleo, no se puede dar por hecho que el planificador va a mantener al *benchmark* casi permanentemente en ejecución.

La segunda opción planteada, sí permite obtener el tiempo de ejecución de un proceso. Para ello, se hace uso de la función *get_proc_stats()* de la biblioteca auxiliar *libproc* [19]. Esta función obtiene en una estructura de tipo *proc_t* la información del proceso, a través del sistema de ficheros del núcleo *procfs* (montado en */proc*). A partir de esta información, es posible calcular el tiempo de ejecución como la suma del tiempo de usuario y el tiempo de núcleo.

```
...
            read(fd_L1DAllRef, &cont_L1DAllRef, sizeof(u64));
```

```
                get_proc_stats(childpid, &p);
                uk_time = p.utime + p.stime;
...
```

Si bien este método es capaz de obtener el tiempo de ejecución del *benchmark*, puede plantear un inconveniente. Al estar trabajando en espacio de usuario, y debido a que la lectura del valor del evento y del tiempo de ejecución del proceso no son una operación atómica, es posible que el planificador expulse al proceso monitorizador entre la ejecución de ambas operaciones. Esto haría que ese muestreo y el inmediatamente posterior, no mostraran una relación tiempo de ejecución / valor del contador correcta.

Finalmente, se descubre un tercer método que soluciona los problemas anteriormente planteados, gracias al estudio más detallado de la interfaz del sistema de contadores de rendimiento de Linux. Para ello, hay que configurar el formato de lectura en la estructura *perf_event_attr*.

```
...
  struct perf_event_attr attr_L1DAllRef =
  {
...
    .read_format = PERF_FORMAT_TOTAL_TIME_RUNNING
  };
...
```

De este modo, es posible obtener mediante una única llamada *read()* el valor del contador y el tiempo de ejecución del proceso.

```
...
             read(fd_L1DAllRef, cont_L1DAllRef, sizeof(u64) * 2);
...
```

Hay que destacar que no se han observado diferencias significativas en los resultados con respecto al método anterior usando un período de muestreo de 100 ms. A continuación se muestra cómo queda el bucle de lectura de valores intermedios implementando este último método.

```
...
    do {
      read(fd_L1DAllRef, cont_L1DAllRef, sizeof(u64) * 2);

      if (write(fd_output, cont_L1DAllRef, sizeof(u64) * 2) != sizeof(u64) * 2) {
        perror("No se pudo escribir en el fichero de salida");
        kill (childpid, SIGTERM);
        exit(EXIT_FAILURE);
      }

      usleep(PERIODO);
      waitpid(childpid, &status, WNOHANG);
    } while (!WIFEXITED(status));
...
```

El tiempo transcurrido entre las muestras tomadas por el padre es aproximado, ya que dependerá del tiempo que pase desde que se despierta y pasa a la cola de preparados hasta que el planificador de procesos lo pase a ejecución.

Una vez se ha terminado, se vuelven a leer los contadores para mostrar por pantalla los valores finales. Esta última lectura no es tenida en cuenta para las estadísticas.

```
...
    read(fd_L1DAllRef, cont_L1DAllRef, sizeof(u64) * 2);
    printf("Valores finales:\n");
    printf(" L1D_ALL_REF: %llu\n", cont_L1DAllRef[1]);

    close(fd_L1DAllRef);
    close(fd_output);

    exit(EXIT_SUCCESS);
  }
}
```

De este modo, ya es posible emplear la aplicación *parser* para pasar los datos al formato que utiliza *gnuplot* y visualizar así los resultados en una gráfica.

4.1.4 Selección y conversión de los datos

Antes de realizar las observaciones, hay que preparar los datos para poder representarlos en una gráfica. De esto se encarga la aplicación *parser*, escrita en *C*.

En un primer momento, se trabajó con una única versión para todos los tiempos de muestreo. En dicha versión, *parser* recibe como parámetro el nombre del fichero que contiene los datos en formato binario. Mediante un bucle, lee cada valor del contador y del tiempo de ejecución.

```
...
  valor_ant = 0;
  tiempo_ant = 0;

  while ( (count = read(from_fd, &valor, sizeof(u64))) > 0
  && (count2 = read(from_fd, &tiempo, sizeof(u64))) > 0) {
...
```

En primer lugar, se comprueba si se dan pares de valores repetidos de tiempo de ejecución y valor del contador y se descartan, teniendo cuidado de no eliminar el primer valor. Los valores repetidos indican que el planificador no ha pasado a ejecución al *benchmark* entre dos muestreos, y no aportan nada en este caso.

```
...
    if (tiempo == tiempo_ant && valor == valor_ant && tiempo > 0)
      continue;
...
```

A continuación, el tiempo de ejecución es convertido de nanosegundos a segundos, y al valor del contador se le resta el inmediatamente anterior para calcular el valor contabilizado durante su tiempo de muestreo. Estos dos valores son mostrados por la salida estándar, en formato de dos columnas.

```
...
    printf("%.5f %llu", tiempo / 1000000000.0, valor - valor_ant);
...
```

Como extra, se han añadido dos valores más que no son utilizados por la gráfica, pero que han servido de ayuda durante la realización del proyecto. Estas dos últimas columnas muestran el valor total acumulado por el contador y la diferencia de tiempo con su valor inmediatamente anterior,

pasado a milisegundos.

```
...
    printf(" %llu %.2f\n", valor, (tiempo - tiempo_ant) / 1000000.0);
...
```

Así, los datos se muestran en un total de 4 columnas.

```
#t(s) #Dif_evento #Total_evento #Dif_t(ms)
0.09803 157012455 157012455 98.03
0.19803 138855760 295868215 100.00
0.29805 101751944 397620159 100.02
0.39805 167272634 564892793 100.00
0.49804 137905323 702798116 99.99
...
```

Después de mostrarse cada fila de datos, se actualizan los valores de *valor_ant* y *tiempo_ant*.

```
...
    valor_ant = valor;
    tiempo_ant = tiempo;
  }
...
```

Esta aplicación funciona bien para las mediciones realizadas con valores de suspensión en el bucle de test_bench de 100 ms y 5 s. Sin embargo, al convertir los datos de la ejecución que emplea un valor de referencia de 1 ms, se observan valores del contador anormalmente altos, que desvirtúan los resultados. Un análisis más en profundidad muestra que hay casos en los que la aplicación de monitorización pasa largos períodos de tiempo con respecto al tiempo de referencia sin recoger datos, lo que provoca que se acumulen picos de valores para la siguiente lectura. Al observar con mayor detenimiento los datos con un valor de referencia de 100 ms y 5 s, se observa que en el caso de los 100 ms también se da, aunque rara vez, de forma aleatoria en alguna medición.

Para mitigar esto, se modifica la aplicación *test_bench* de modo que el proceso de monitorización se planifique como proceso de tiempo real, como pudo verse en el punto 4.1.3. También se implementa una segunda versión de la aplicación de conversión, adaptada específicamente para los datos recogidos con un valor de referencia de 1 ms y 100 ms.

En esta versión, la diferencia se encuentra en que antes de imprimir los datos, se comprueba si el tiempo de ejecución del *benchmark* durante el período de muestreo ha superado un cierto umbral. En caso afirmativo, se introduce un comentario al principio de la línea. Esto evita que dicho dato sea mostrado posteriormente en la gráfica, pero permitiendo realizar observaciones fuera de la misma.

```
...
    #ifndef T_5S
    if (tiempo - tiempo_ant > UMBRAL)
      printf("# ");
    #endif
...
```

El valor del umbral escogido tras realizar varias observaciones ha sido de un 50% sobre el valor de referencia, que es de 1'5 ms para 1 ms y 150 ms para 100 ms. El objetivo es obtener un equilibrio

entre datos descartados y valores corrientes, ya que el período de muestreo real suele estar ligeramente por encima del valor de referencia en el caso de 1 ms. Este valor probablemente podría ser afinado con un análisis estadístico más detallado.

```
...
#ifdef T_1MS
  #define UMBRAL 1500000
#elif T_100MS
  #define UMBRAL 150000000
...
```

Debido a que los datos se muestran por la salida estándar, es preciso redirigir la salida a un fichero durante la ejecución, para poder después trabajar con ellos.

4.1.5 Representación de la información

Para representar la información, se utilizan gráficas de dos dimensiones. En el plano, el eje de abscisas representa el tiempo de ejecución del *benchmark*, mientras que el eje de ordenadas muestra el número de eventos producidos durante el período de muestreo.

En el ejemplo de la figura 25, puede verse la representación del *benchmark 403.gcc* en una fase anterior del proyecto, en la que se estuvo estudiando cómo trabajar con más de un evento de forma simultánea. El tiempo de referencia es de 100 ms.

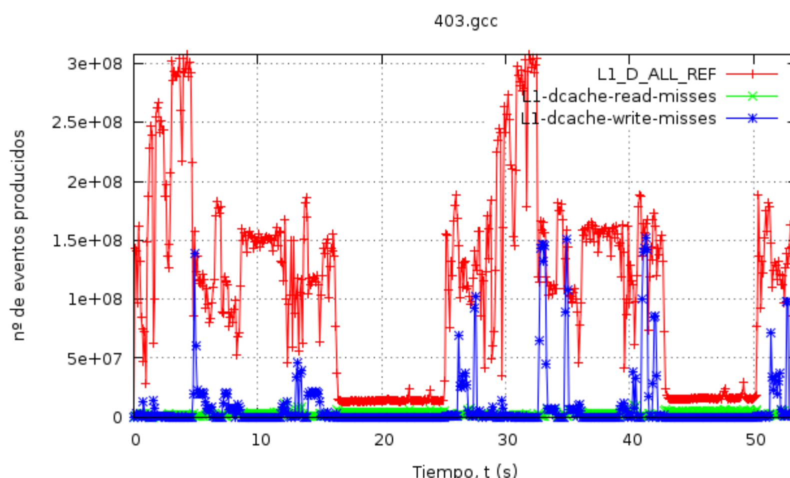

Figura 25: Ejemplo de ejecución de un benchmark con 3 eventos diferentes

El programa empleado para este fin es *gnuplot*. Para poder utilizarlo simplemente hay que indicar el fichero de configuración *.plt* previamente generado por el guión *runbench*.

```
$ gnuplot -p 403.gcc.plt
```

En este fichero de configuración se indican varios parámetros como el título, el nombre de los ejes o el fichero de datos que se ha de leer junto con la relación eje – columna de datos que se desea mostrar.

```
set title "403.gcc"
set xlabel "Tiempo, t (s)"
set ylabel "nº de eventos producidos"
set grid
set autoscale fix
plot \
"403.gcc.dat" using 1:2 title "L1_D_ALL_REF" with linespoints, \
"403.gcc.dat" using 1:3 title "L1-dcache-read-misses" with linespoints, \
"403.gcc.dat" using 1:4 title "L1-dcache-write-misses" with linespoints
```

También es posible configurar *gnuplot* para generar ficheros de imágenes, en lugar de mostrar directamente la gráfica. Para ello hay que añadir dos opciones extra justo antes de la opción *plot*, indicando las propiedades de la imagen y el fichero de salida.

```
...
set terminal pngcairo size 1272,384 font "LiberationSans-Regular,10"
set output "403.gcc.png"
...
```

Este paso también ha sido automatizado por medio del guión *runbench*.

4.2 Realización de las pruebas de rendimiento

En esta sección se comentan los preparativos del entorno de pruebas, y a continuación se muestran y explican de forma detallada los resultados finales de la ejecución de cada uno de los *benchmarks*.

4.2.1 Configuración del entorno de ejecución

Para la realización de las pruebas de rendimiento, se ha utilizado una distribución GNU/Linux con un núcleo 2.6.39.2 compilado para la arquitectura *i686*, a pesar de que el procesador soporta la arquitectura *x86_64*. Por otro lado, los *benchmarks* están precompilados para la arquitectura *i386*, tal y como se ofrecen en la página del proyecto *Multi2sim* [18].

Todas las pruebas han sido realizadas en modo monousuario. En el modo monousuario de Unix, no se arrancan los demonios de inicio. De este modo, el entorno de ejecución se puede mantener con una carga mínima, es decir, sin aplicaciones extra que puedan desvirtuar los resultados.

Los tiempos de ejecución de algunos *benchmarks* pueden pasar de la media hora. Esto hizo que fuera necesario escribir un pequeño guión en *Bash* que permitiera la realización de las pruebas de forma desatendida. Este guión se encarga de lanzar de forma secuencial instancias del guión *runbench* con cada uno de los *benchmarks* disponibles, pasando la opción *--no-plot* para que no se muestre la gráfica en el *gnuplot* tras la ejecución de cada uno de ellos. A través del código de finalización de cada prueba, se escribe en un fichero de diario si la ejecución de cada una de ellas ha sido correcta. Esto permite posteriormente comprobar si ha habido algún problema.

```
...
for bench in $BENCH_LIST; do
  for _periodo in $_PRUEBAS; do
    export _periodo
    runbench --no-plot $bench
    if [ $? -eq 0 ]; then
      logline="$(date "+%F %T"): ${bench}${_periodo} -> OK"
    else
      logline="$(date "+%F %T"): ${bench}${_periodo} -> ERROR"
    fi
    mv /tmp/bench_output.data $RES_DIR/${bench}${_periodo}.data
    echo "$logline" >> "$LOG_FILE"
  done
done
```

Cabe recordar que los parámetros de ejecución concretos para cada *benchmark* son los indicados por el proyecto *Mult2Sim* [18].

4.2.2 Resultados de la ejecución de los benchmarks

A continuación se muestran los resultados de la ejecución de 10 pruebas. 5 de ellas emplean *benchmarks* de enteros y las otras 5 *benchmarks* de coma flotante, de entre los 31 ofrecidos por *CPU2006*. Todas las pruebas han sido realizadas con el evento *L1D_ALL_REF* y utilizando períodos de muestreo de 1 ms, 100 ms y 5 s.

Benchmarks de enteros

400.perlbench

Parámetros: “`checkspam.pl 2500 5 25 11 150 1 1 1 1`”

En la ejecución de este *benchmark*, se puede observar que el número de accesos a memoria varía de forma importante al principio y luego mantiene un comportamiento de fluctuaciones más o menos acotadas durante el resto de la ejecución.

En la figura 26 pueden verse los resultados con un período de 1 ms. La mayor parte de los valores se mantiene entre $1'2 \cdot 10^6$ y $2'5 \cdot 10^6$, si bien hay un gran número de valores que fluctúa fuera de estos límites.

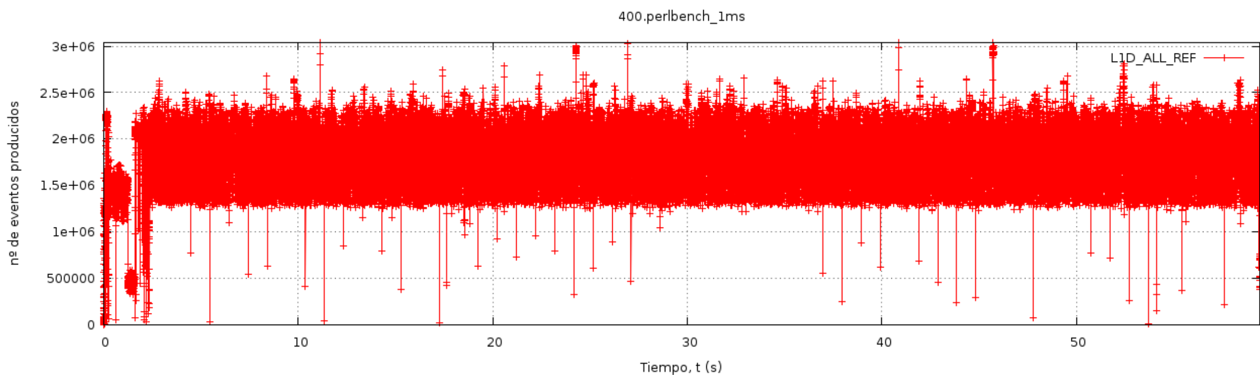

Figura 26: 400.perlbench, 1 ms

Con un período de 100 ms, una vez los accesos a memoria se estabilizan, se mantienen entre $1'6 \cdot 10^8$ y $1'9 \cdot 10^8$. En la figura 27 se puede observar cómo las fluctuaciones son menos agresivas que en el caso del período de 1 ms. Como el período es 100 veces mayor, los valores están en el orden de 10^8, en lugar de 10^6.

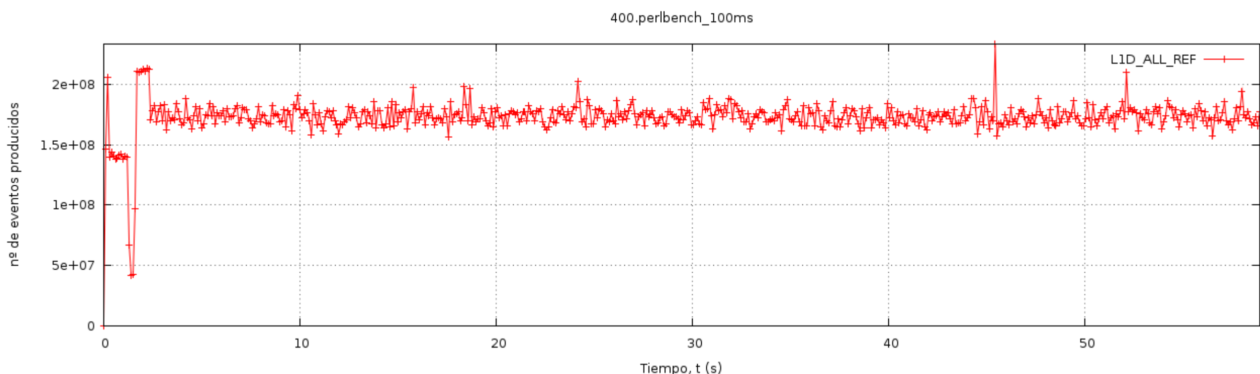

Figura 27: 400.perlbench, 100 ms

En el caso del período de 5 s, los accesos a memoria crecen rápidamente hasta superar los $8 \cdot 10^9$ y luego mantienen una línea entre los $8'6 \cdot 10^9$ y $8'8 \cdot 10^9$ accesos, como puede verse en la figura 28.

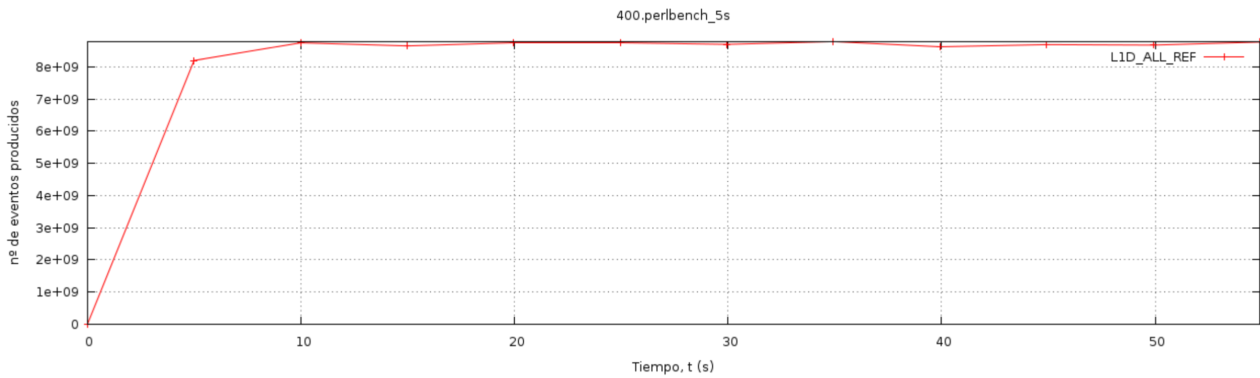

Figura 28: 400.perlbench, 5 s

403.gcc

Parámetros: “`166.i -o 166.s`”

En este *benchmark*, se puede observar claramente que existe un patrón en el comportamiento de los accesos a memoria, que se repite en dos ocasiones a lo largo de su ejecución.

En la figura 29, se puede ver cómo con un período de 1 ms los accesos a memoria se comportan de forma más agresiva.

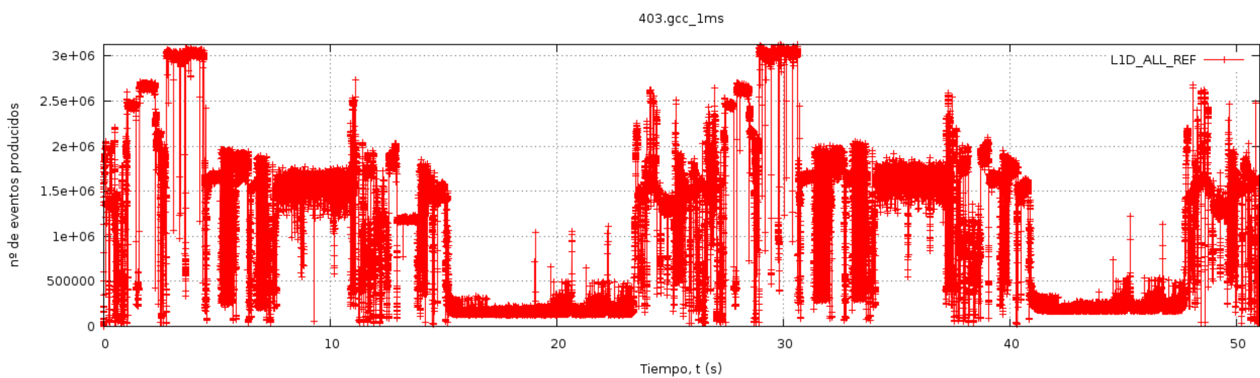

Figura 29: 403.gcc, 1 ms

La figura 30 muestra los resultados con un período de 100 ms. En este caso puede observarse un comportamiento más suave, ya que al emplear un período superior es menos sensible a los cambios bruscos.

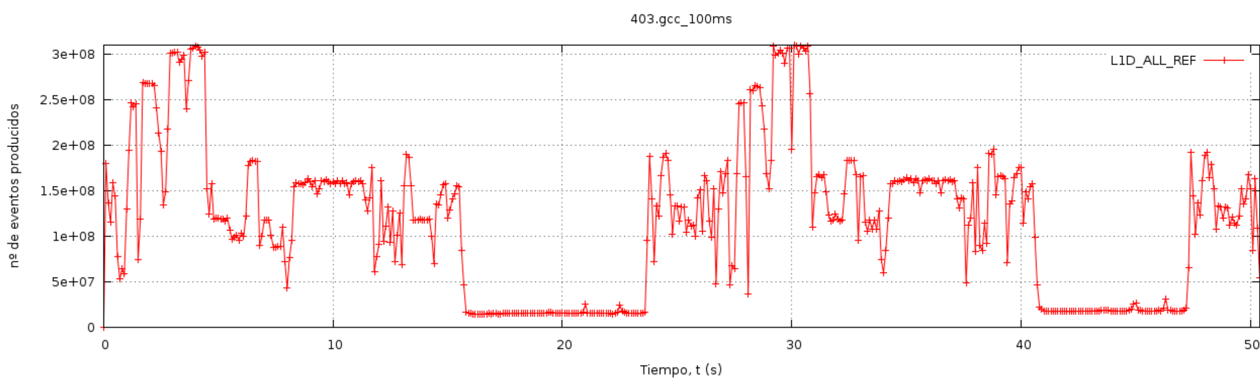

Figura 30: 403.gcc, 100 ms

Con un período de 5 s, se dispone de un número más limitado de muestreos, pero se puede advertir un comportamiento similar en cuanto a los accesos a memoria (figura 31).

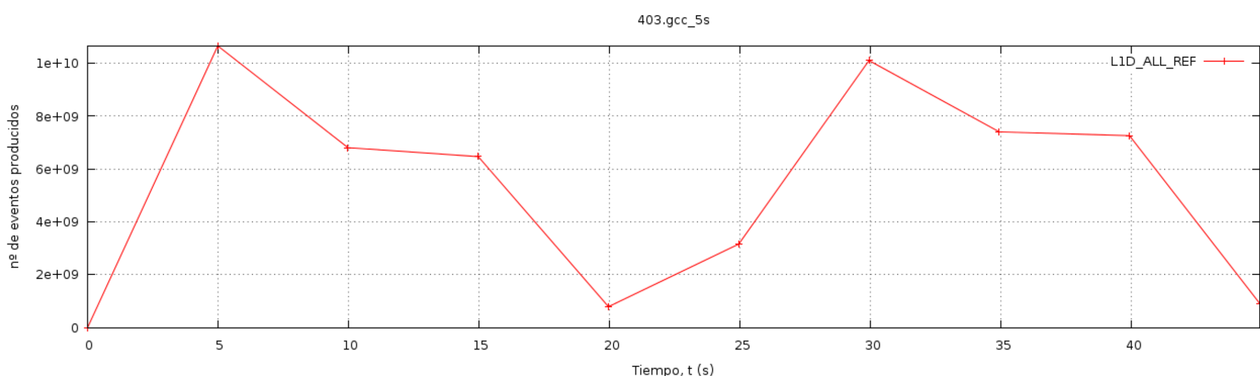

Figura 31: 403.gcc, 5 s

429.mcf

Parámetros: “`inp.in`”

En este *benchmark*, se aprecia cómo se sigue un patrón en el que hay una zona más corta con una mayor cantidad de accesos a memoria, y a continuación le sigue una zona más larga en la que los requisitos de acceso a memoria son menores.

En la figura 32, con un período de 1 ms, pueden observarse cambios muy agresivos en los accesos a memoria de la aplicación. Esto puede apreciarse especialmente en la zona más corta.

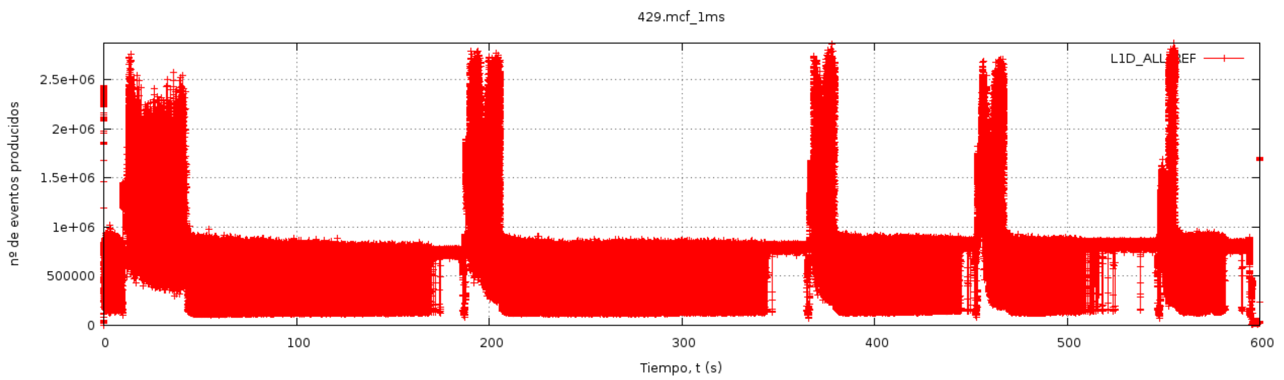

Figura 32: 429.mcf, 1 ms

En la figura 33, se toma un período de 100 ms. Se puede seguir observando el patrón, pero de forma más suave. En la zona corta, ahora pueden diferencia dos niveles de accesos a memoria, con una transición entre ambos apreciable en la gráfica.

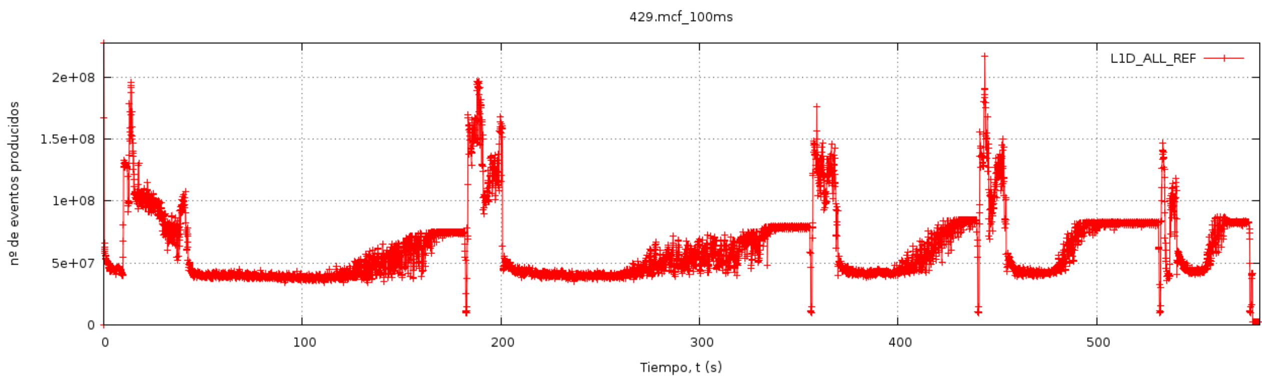

Figura 33: 429.mcf, 100 ms

En la figura 34 puede verse cómo si se aumenta el período de referencia a 5 s, el patrón aprece con un comportamiento mucho menos agresivo.

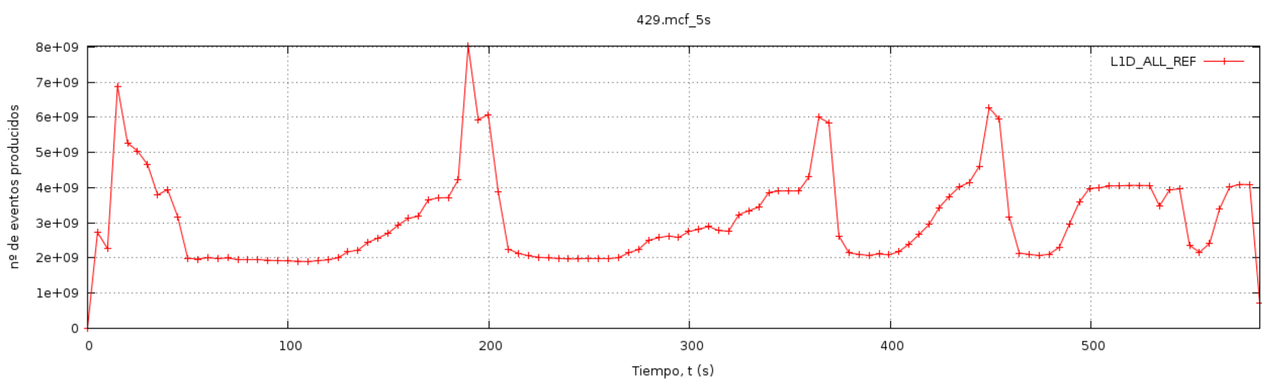

Figura 34: 429.mcf, 5 s

473.astar

Parámetros: “`rivers.cfg`”

Este *benchmark* muestra un comportamiento de accesos a memoria por etapas. En total, pueden diferenciarse entre 5 y 6 etapas, dependiendo del período empleado.

La figura 35 muestra los resultados con un período de 1 ms. A simple vista pueden distinguirse 5 etapas, en las que los accesos a memoria tienen fluctuaciones de diferente grado. Entre la 3ª y la 4ª etapa aparece también una pequeña zona de aproximadamente 1'4 segundos con fluctuaciones importantes.

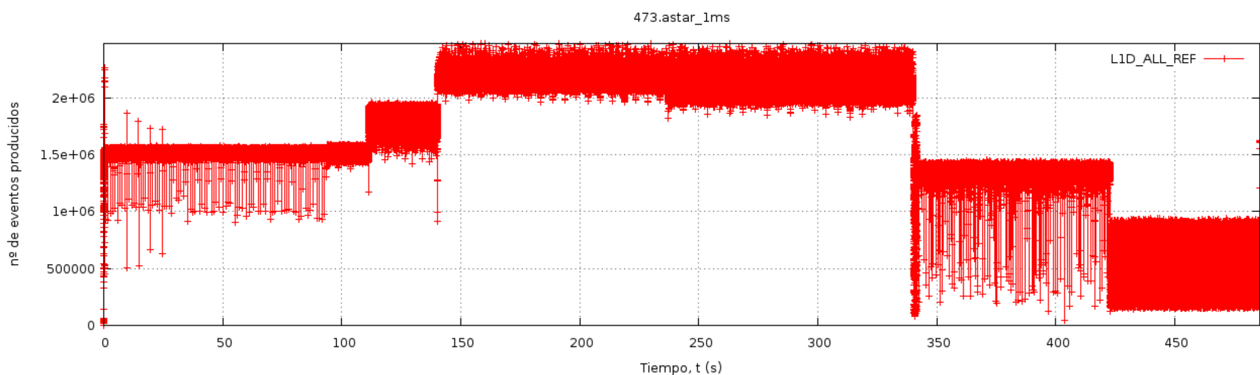

Figura 35: 473.astar, 1 ms

En la figura 36 se muestran los resultados con un período de 100 ms. En este caso, se puede observar que la 3ª etapa realmente está dividida en dos partes.

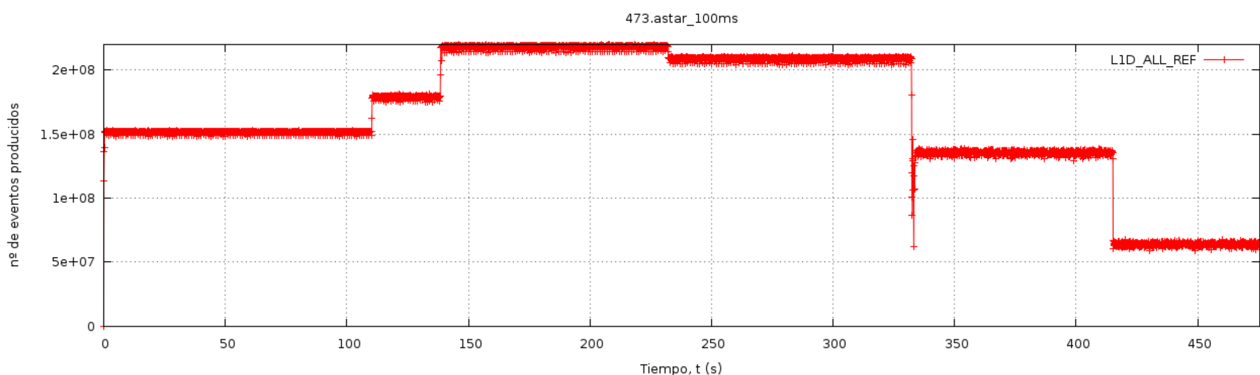

Figura 36: 473.astar, 100 ms

Finalmente en la figura 37 se observa la evolución de los accesos a memoria con un período de 5 s. Los resultados muestran las mismas etapas que aparecen en la gráfica con un período de 100 ms.

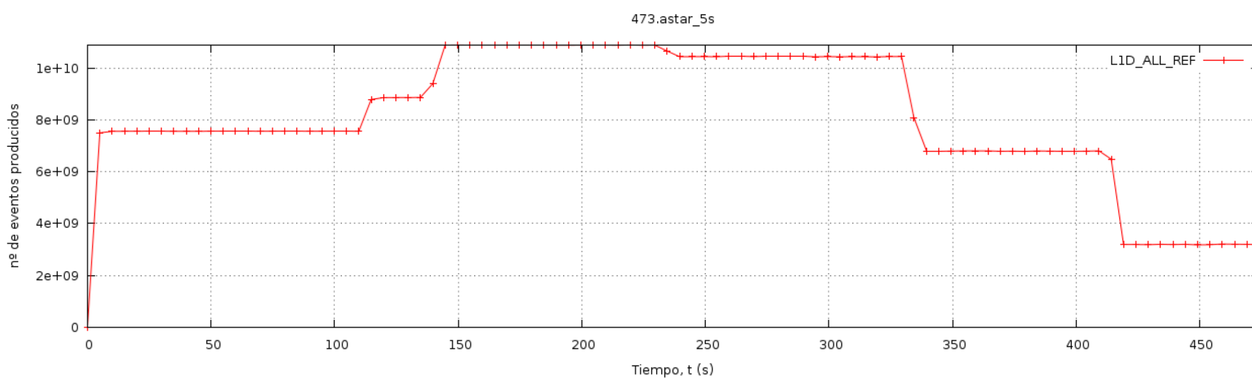

Figura 37: 473.astar, 5 s

483.xalancbmk

Parámetros: “`-v t5.xml xalanc.xsl`”

El comportamiento de accesos a memoria en este *benchmark* puede dividirse en 3 partes.

Con un período de 1 ms, los accesos a memoria se vuelven muy agresivos. En la figura 38, se puede observar que en la primera zona los accesos a memoria se muestran irregulares. Después se pasa a una segunda zona con un nivel de accesos a memoria más alto. Por último, el resto de la ejecución se muestra con accesos a memoria muy cambiantes, aunque con la inmensa mayoría de sus valores acotados entre 0 y 2'2·10^6 eventos.

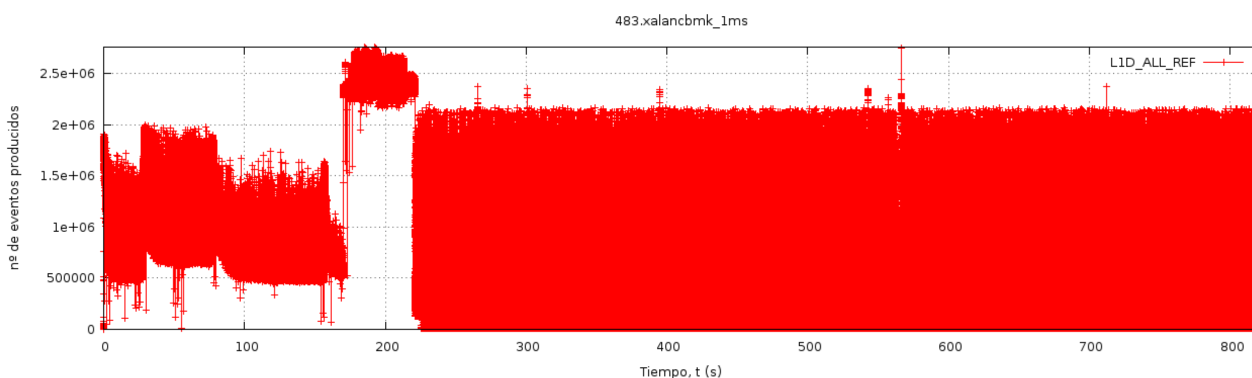

Figura 38: 483.xalancbmk, 1 ms

La figura 39 muestra los resultados con un período de 100 ms. En la primera zona, se observa cómo se repite un patrón en el que los accesos a memoria crecen rápidamente para después pasar a decrecer en forma de curva. Después de esta zona, aparece otra zona con gran cantidad de accesos a memoria. La última zona, muestra importantes fluctuaciones, aunque permite ver cómo los accesos a memoria crecen lentamente a lo largo del resto de la ejecución.

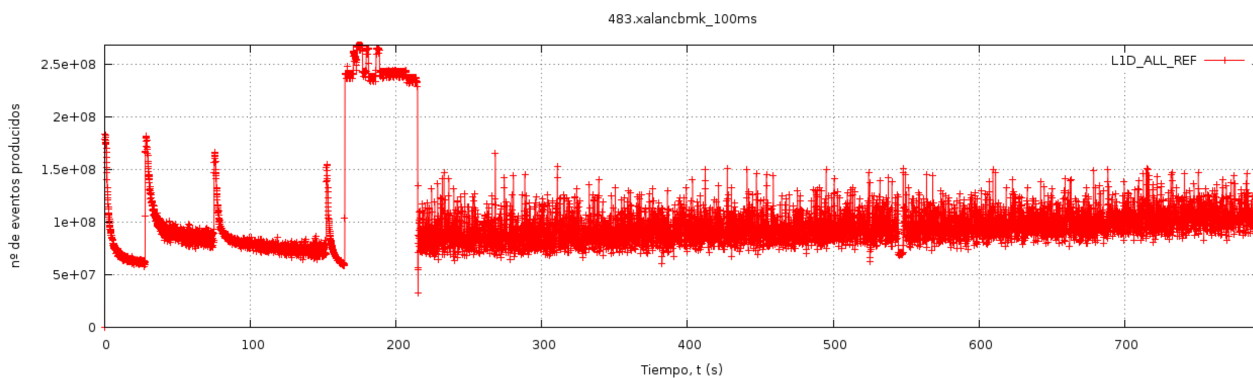

Figura 39: 483.xalancbmk, 100 ms

La figura 40 muestra la gráfica con un período de 5 s. Los resultados se muestran de forma más suave que en la gráfica de 100 ms. Se puede observar también la línea que sigue la pendiente de la última zona, sin fluctuaciones pronunciadas.

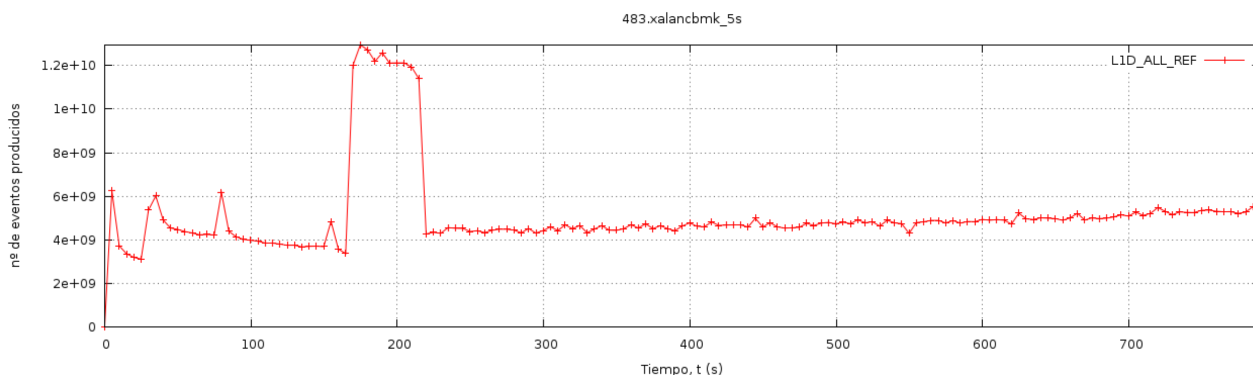

Figura 40: 483.xalancbmk, 5 s

Benchmarks de coma flotante

436.cactusADM

Parámetros: “`benchADM.par`”

El tiempo de ejecución de este *benchmark* es elevado, con una media superior a los 34 minutos. Al comienzo crece drásticamente el número de accesos a memoria, para luego reducirse rápidamente y volver a aumentar lentamente durante el resto de la ejecución.

Con un período de 1 ms, el número de muestreos supera los 2 millones. Además, los accesos a memoria se muestran muy agresivos. Debido a esto, resulta complicado apreciar los detalles en la figura 41.

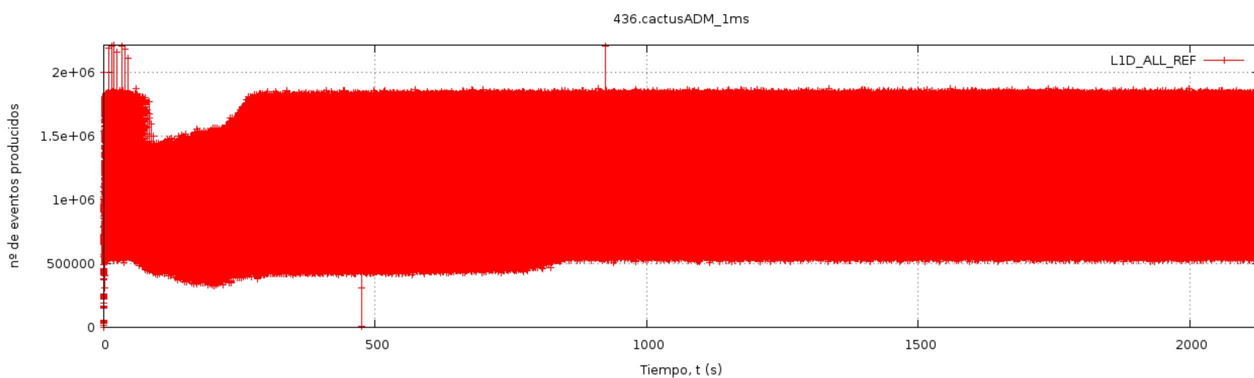

Figura 41: 436.cactusADM, 1 ms

A continuación se ofrecen los resultados empleando un período de 100 ms. En la figura 42 puede apreciarse el comportamiento de los accesos a memoria. A lo largo de la ejecución, las fluctuaciones disminuyen a medida que aumenta la tasa de accesos a memoria.

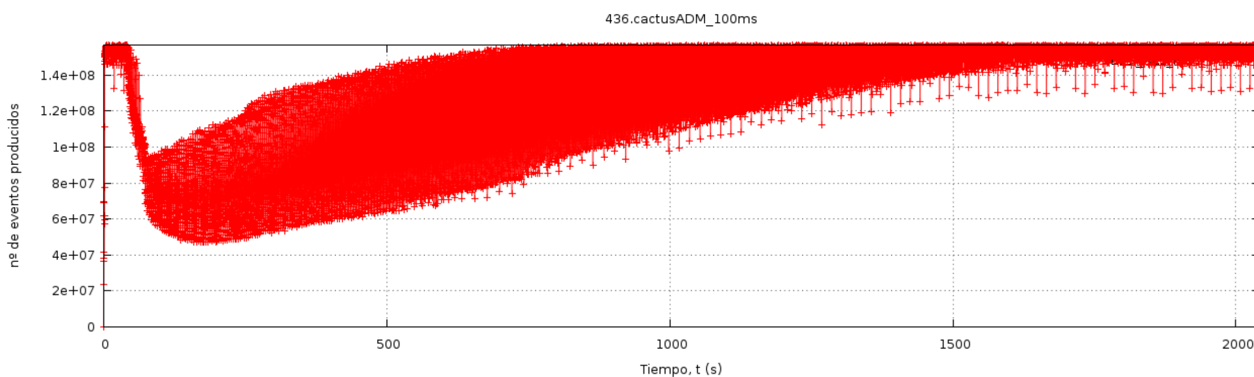

Figura 42: 436.cactusADM, 100 ms

En la figura 43 puede observarse la gráfica con un período de 5 s. El progreso de los accesos a memoria se encuentra plasmado de forma más clara.

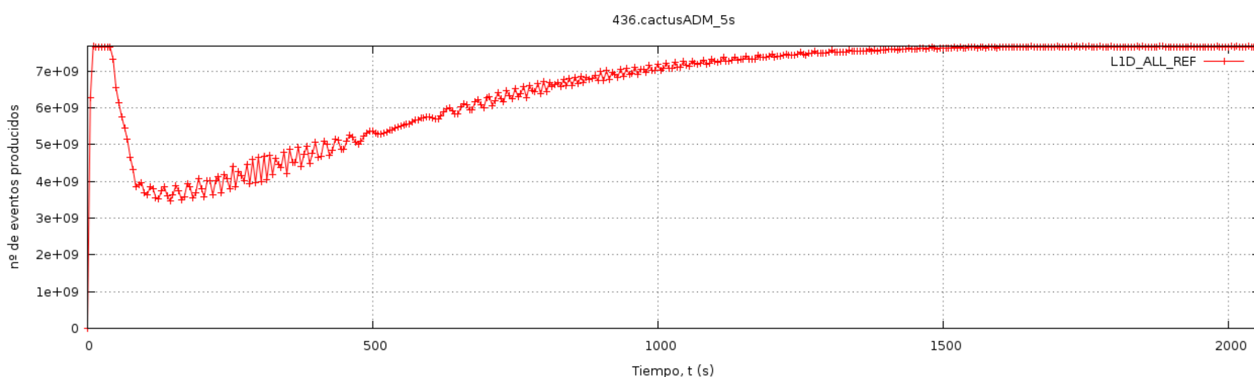

Figura 43: 436.cactusADM, 5 s

444.namd

Parámetros: “`--input namd.input --iterations 38 --output namd.out`”

En este *benchmark*, puede observarse claramente un patrón de accesos de unos 21 segundos, que se repite a lo largo de toda la ejecución.

La figura 44 muestra la gráfica con un período de 1 ms, en la que el patrón se repite en 38 ocasiones.

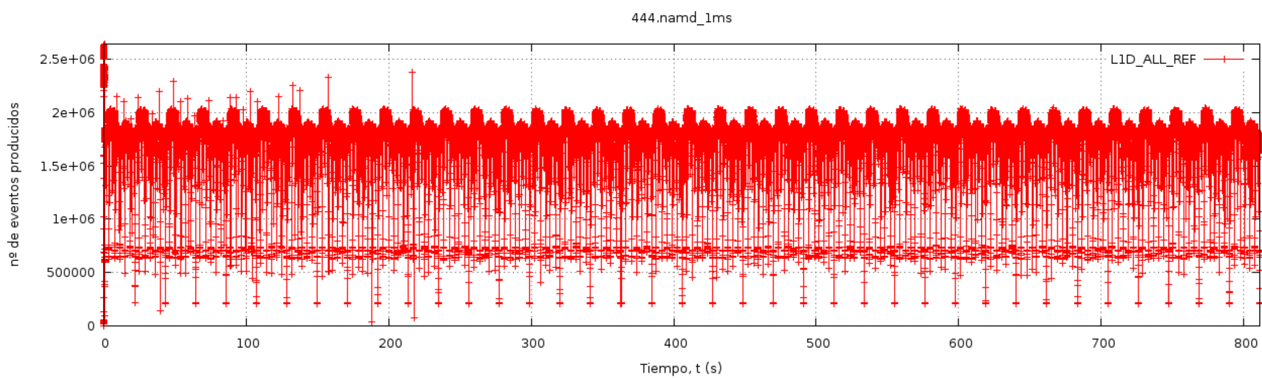

Figura 44: 444.namd, 1 ms

En la figura 45 se emplea un período de 100 ms, y también puede observarse cómo el patrón de accesos a memoria se repite 38 veces.

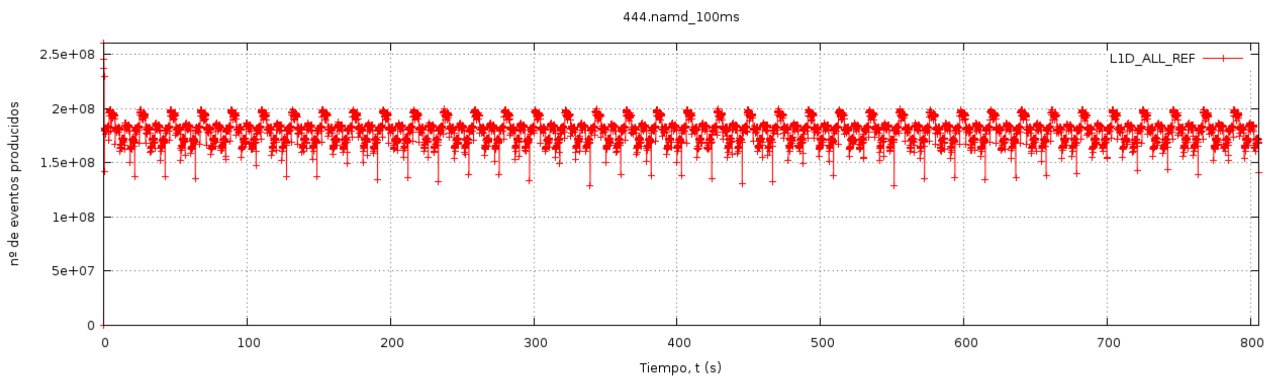

Figura 45: 444.namd, 100 ms

En la figura 46 se muestran los resultados empleando un período de 5 s. Un período de muestreo tan grande comparado con el tamaño del patrón, hace que sea complicado identificarlo.

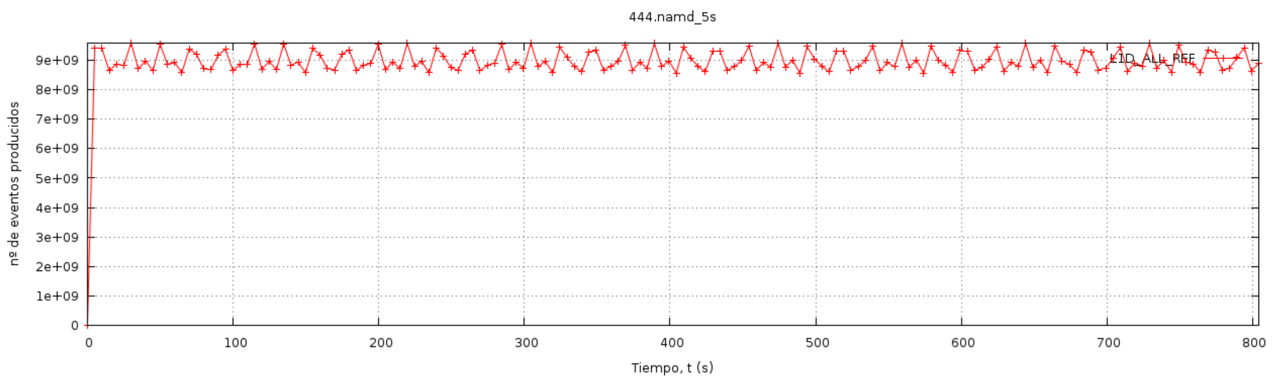

Figura 46: 444.namd, 5 s

447.dealII

Parámetros: “23”

En este *benchmark* de coma flotante, aparece un patrón de accesos a memoria con un período creciente a lo largo de toda su ejecución.

En la figura 47 se puede observar este patrón. Con un tiempo de ejecución de 1 ms, los accesos a memoria ofrecen fluctuaciones que mantienen un comportamiento similar en cada aparición del patrón.

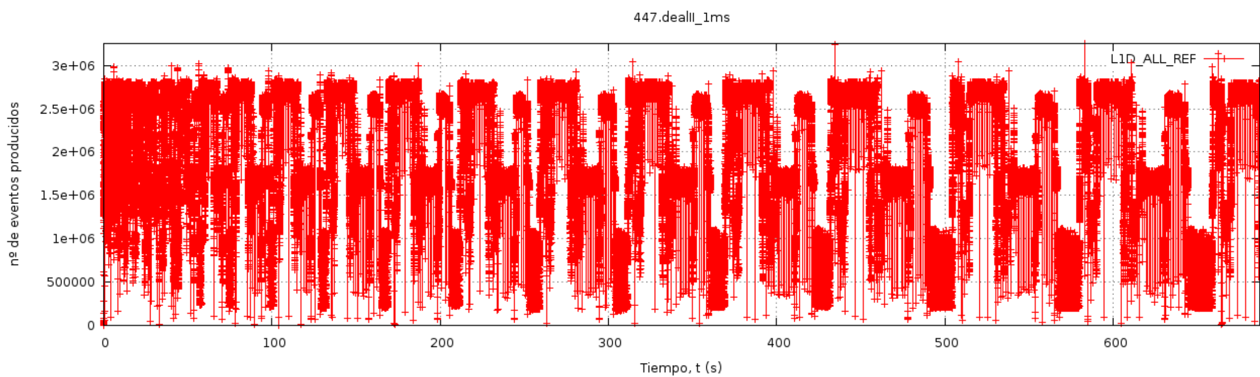

Figura 47: 447.dealII, 1 ms

Con un período de 100 ms, el patrón puede verse de forma más clara en la figura 48, ya que las fluctuaciones son menores.

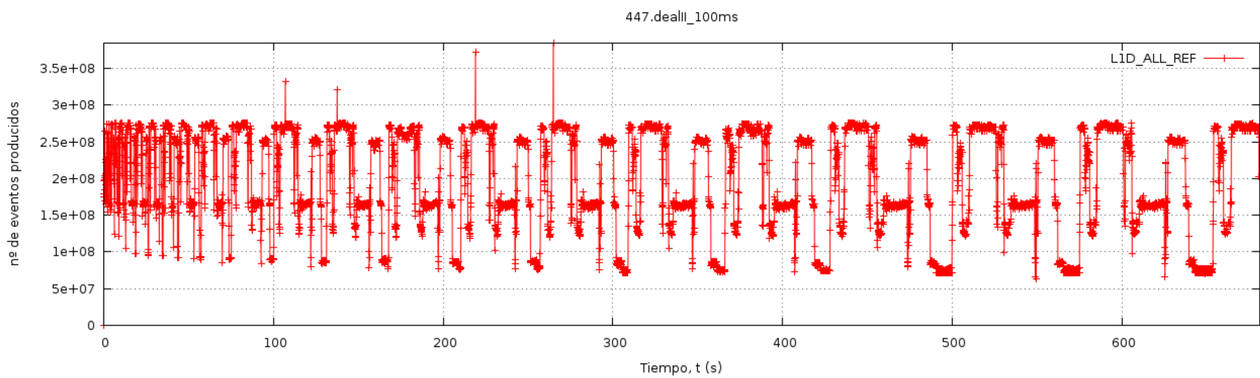

Figura 48: 447.dealII, 100 ms

En la figura 49, se muestran los resultados con un período de 5 s. El patrón se puede seguir observando, sin embargo sólo puede distinguirse con claridad cuando el período del mismo aumenta lo suficiente.

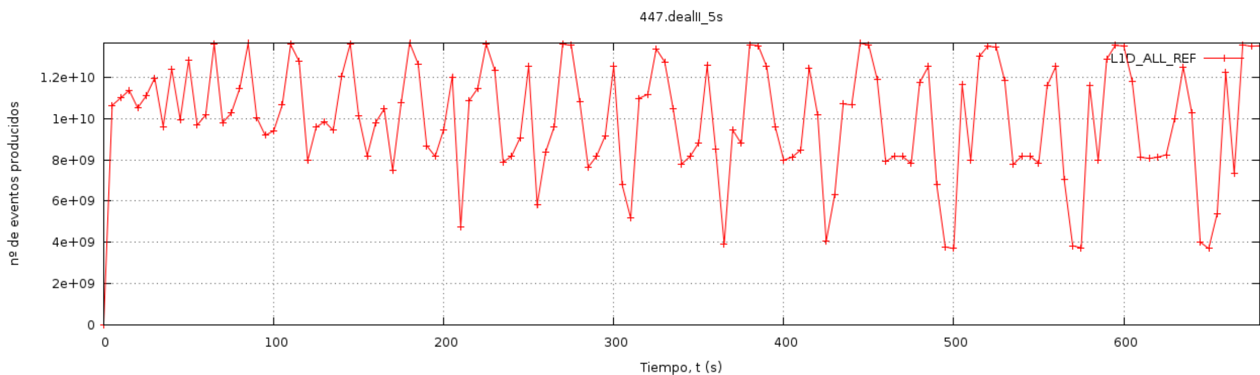

Figura 49: 447.dealII, 5 s

450.soplex

Parámetros: "`-m3500 ref.mps`"

En este *benchmark*, los accesos a memoria son irregulares al principio de la ejecución, para luego pasar a una zona con importantes cambios muy rápidos en el número de accesos a memoria, cuya media se mantiene estable.

En la figura 50, puede verse el resultado tras realizar la ejecución con un período de 1 ms. Los accesos a memoria mantienen un comportamiento muy agresivo durante toda la ejecución. Es posible distinguir los cambios que hay en la zona inicial del resto de la ejecución.

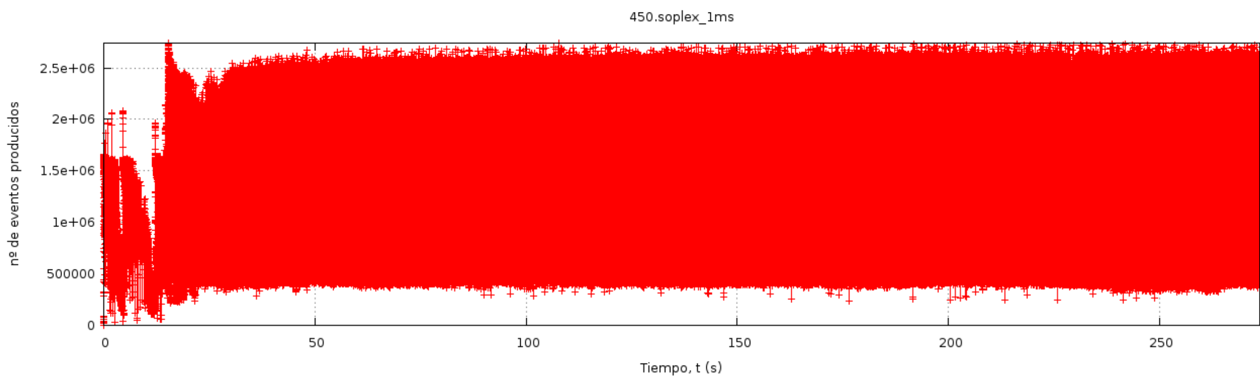

Figura 50: 450.soplex, 1 ms

La figura 51 muestra la ejecución con un período de 100 ms. La zona inicial con diferentes requisitos de accesos a memoria ahora puede distinguirse con mucha más claridad, mientras que el resto de la ejecución muestra una línea que se mantiene más o menos estable, dentro de las fluctuaciones que se producen a lo largo de la misma.

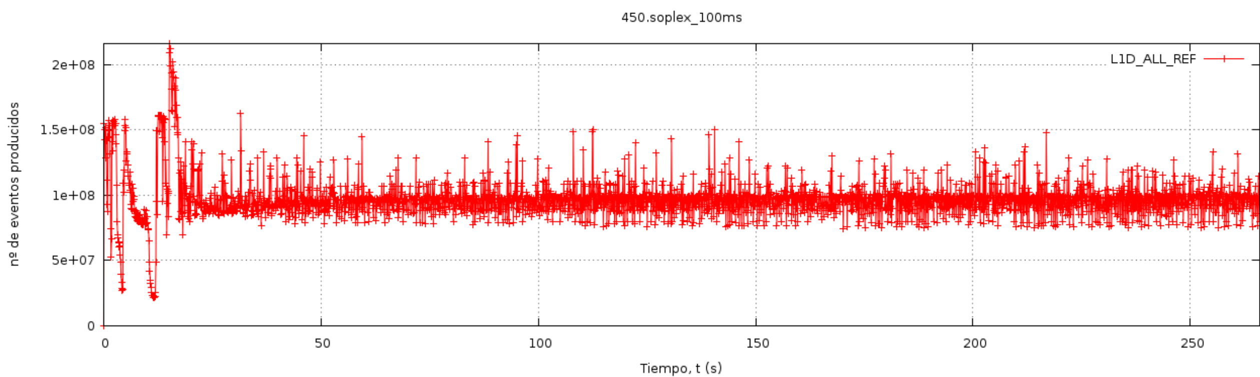

Figura 51: 450.soplex, 100 ms

En la figura 52 se muestran los accesos a memoria con un período de 5 s. Aquí se puede distinguir la zona inicial, con dos picos importantes de accesos a memoria, mientras que la línea estable de accesos a memoria puede verse mucho más suave.

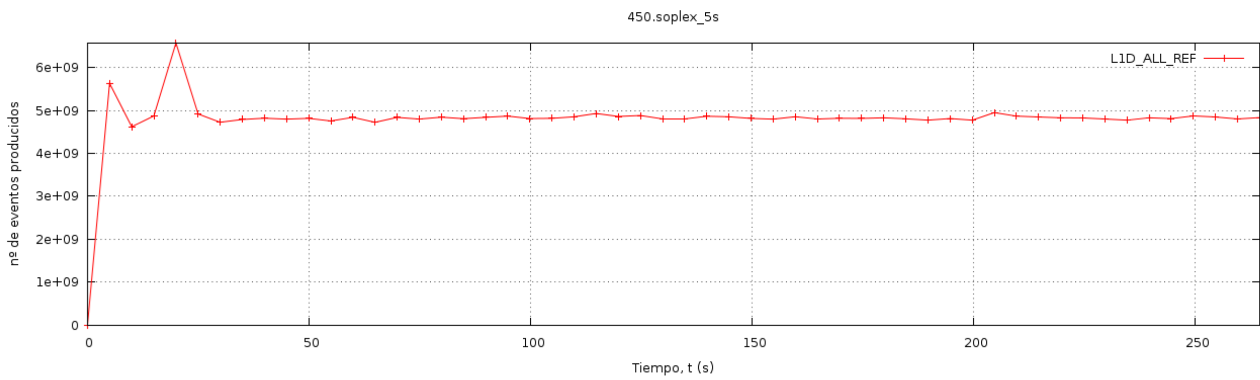

Figura 52: 450.soplex, 5 s

465.tonto

Parámetros: ninguno.

El último *benchmark* de coma flotante, muestra un patrón que se repite en 8 ocasiones a lo largo de la ejecución, tras una zona inicial en la que los accesos a memoria tienen una media alta. Por otro lado, el patrón se divide en una zona con un alto número de accesos a memoria y en otra zona con un menor número de accesos.

La figura 53 muestra los accesos a memoria haciendo uso de un período de 1 ms. La zona inicial muestra las importantes fluctuaciones que se producen en los requisitos de accesos a memoria. Curiosamente, la repetición del segundo patrón también tiene una zona con un comportamiento más agresivo, que lo diferencia del resto de repeticiones.

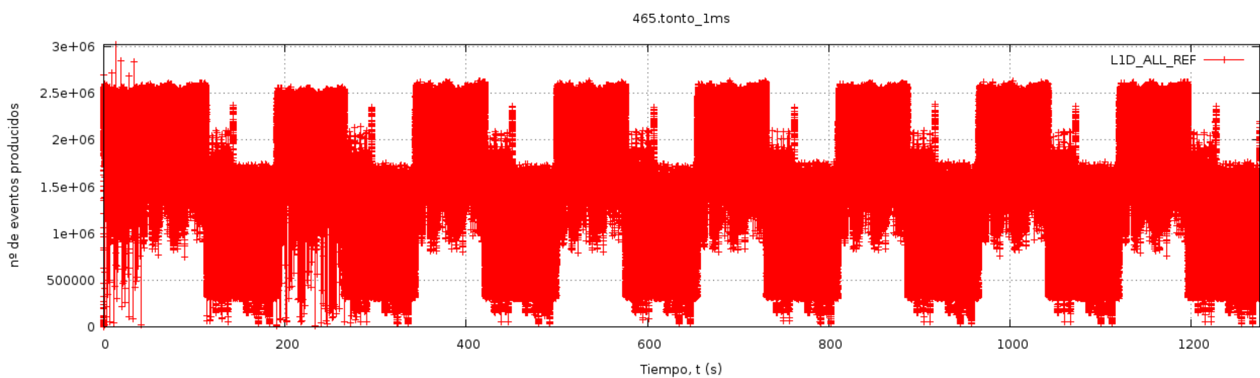

Figura 53: 465.tonto, 1 ms

En la figura 54, con un período de 100 ms, se pueden observar las zonas alta y baja del patrón con un aspecto más definido, ya que las fluctuaciones son menores.

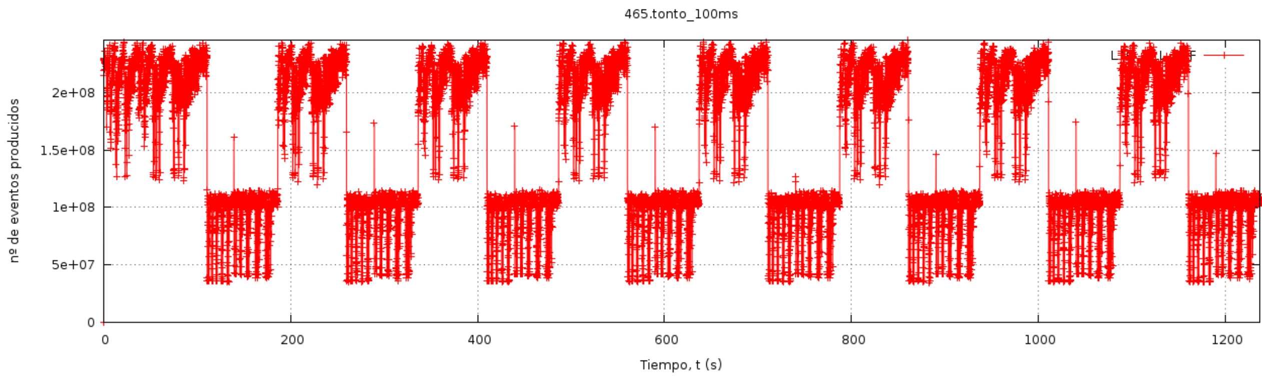

Figura 54: 465.tonto, 100 ms

Finalmente, en la figura 55 se muestran los resultados empleando un período de 5 s. El patrón se muestra en este caso con un trazo más sencillo.

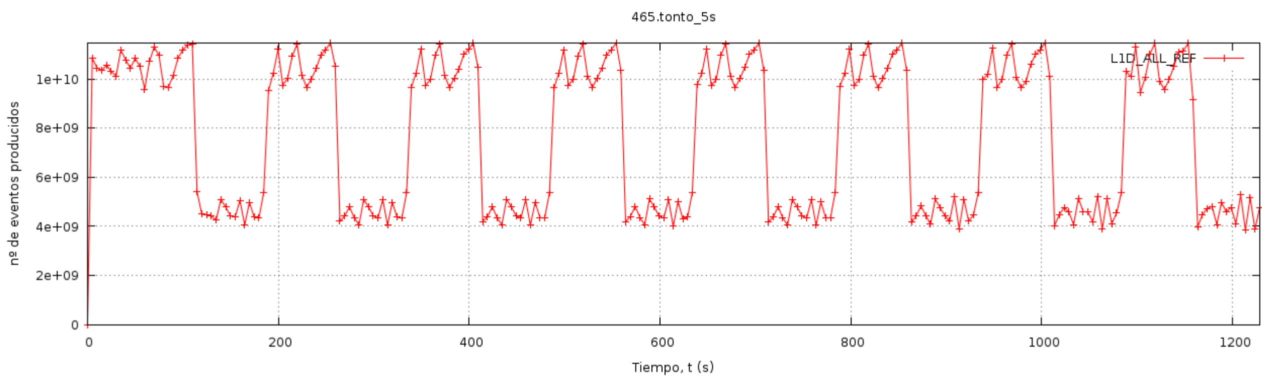

Figura 55: 465.tonto, 5 s

5 Conclusiones y trabajo futuro

El trabajo realizado en este proyecto, muestra una posible forma de realizar la caracterización de aplicaciones. Esta caracterización se confecciona fuera de línea, es decir, los resultados pueden ser analizados en una gráfica una vez finalizada la ejecución de cada prueba.

A la hora de seleccionar el evento para monitorizar, ha habido que consultar y utilizar un evento concreto ofrecido por la microarquitectura en la que está basada el procesador con el que se han realizado las experiencias. Esto dificulta la portabilidad con procesadores de otras microarquitecturas o fabricantes. La implementación de los contadores de rendimiento del sistema operativo, ofrece una abstracción para determinados tipos de eventos. Lamentablemente, el evento que se ha utilizado en este proyecto, no se encuentra disponible en esta lista de eventos generalizada.

Durante el desarrollo de la herramienta, se han encontrado problemas para que el período de monitorización se mantuviera estable. Esto es debido a que el proceso monitorizador es un proceso más que debe competir por la CPU. Se ha tratado de minimizar este efecto aumentando la prioridad del proceso y ajustando la aplicación de tratamiento de datos. Sin embargo, para lograr una mayor precisión, sería conveniente realizar la monitorización desde el propio núcleo.

Es posible observar que cuanto menor es el período de muestreo, se producen cambios más bruscos en los requisitos de accesos a memoria de la aplicación, que generan picos y caídas. Los picos altos son muy importantes, ya que pueden alcanzar el límite de la capacidad de la memoria, pasando desapercibidos cuando se utilizan tiempos de referencia superiores. Esto debería ser estudiado con mayor detenimiento en un trabajo futuro.

Como trabajo futuro se podría realizar un planificador de procesos que tome decisiones guiadas por el comportamiento de las aplicaciones, es necesario hacer la caracterización en tiempo real. Para ello, el planificador deberá emplear una ventana de tiempo en la que recoja la evolución de los accesos a memoria o la utilización del ancho de banda del bus de memoria. Tras aplicar un algoritmo que haga uso de esta información, tendrá que decidir qué grupos de tareas deben pasar a ejecución en cada *quantum* de tiempo de planificación.

Otra alternativa de planificador de procesos que se podría implementar sería un planificador que utilizara la información que haya sido obtenida anteriormente de la ejecución de las aplicaciones. Este planificador conocería de antemano el comportamiento futuro de la aplicación, con lo que podría mejorar las decisiones de planificación.

El método para caracterizar las aplicaciones empleado en este proyecto, es extensible a otros tipos de evento. En este caso, se ha utilizado un evento para caracterizar las aplicaciones en cuanto a accesos a memoria, pero en un trabajo futuro habría que estudiar qué otros eventos pueden ser de utilidad, como el ancho de banda del bus de memoria utilizado.

Bibliografía

[1] C. D. Antonopoulos, D. S. Nikolopoulos y T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs.

[2] Di Xu, Chenggang Wu y Pen-Chung Yew. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling.

[3] Intel Core microarchitecture. Wikipedia, the free enciclopedia.
http://en.wikipedia.org/wiki/Intel_Core_%28microarchitecture%29

[4] Inside Intel Core microarchitecture.
ftp://download.intel.com/technology/architecture/new_architecture_06.pdf

[5] Intel(R) 64 and IA-32 architectures software developer's manual, volume 3B.
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[6] Nehalem performance monitoring unit guide.
http://software.intel.com/file/30388

[7] Intel processor identification and the CPUID instruction.
http://www.intel.com/content/dam/doc/application-note/processor-identification-cpuid-instruction-note.pdf

[8] Dueling performance monitors.
http://lwn.net/Articles/310260/

[9] Followups: performance counters, ksplice, and fsnotify.
http://lwn.net/Articles/311850/

[10] tools/perf/design.txt dentro de la rama principal de Linux 2.6.39.

[11] Stéphane Eranian. En la CScADS Summer Workshops 2009.
http://cscads.rice.edu/workshops/summer09/slides/performance-tools/cscads09-eranian.pdf

[12] Stéphane Eranian. En la CScADS Summer Workshops 2010.
http://cscads.rice.edu/workshops/summer-2010/slides/performance-tools/perf_events_status_update.pdf

[13] Vince Weaver. The Unofficial Linux Perf Events Web-Page.
http://web.eecs.utk.edu/~vweaver1/projects/perf-events/index.html

[14] Sitio web de perfmon2.
http://perfmon2.sourceforge.net/

[15] Sitio web del SPEC.
http://www.spec.org/

[16] John L. Henning. SPEC CPU2006 Memory Footprint.

[17] SPEC CPU Subcommittee and the original program authors. SPEC CPU 2006 Benchmark Descriptions.

[18] Sitio web de Multi2Sim.
http://www.multi2sim.org/wiki/index.php5/SPEC2006_Execution_Commands

[19] Sitio web de PROCPS.

http://procps.sourceforge.net/

[20] Sitio web de Gnuplot.
http://www.gnuplot.info/